Scheme 语言简明教程

wizardforcel

Published
with GitBook



目錄

Scheme语言简明教程

> 译者：songjinghe
>
> 来源：TYS-zh-translation

译:Teach Yourself Scheme in Fixnum Days

View On GitHub | 下载中文翻译ZIP | 英文原文online | 反馈问题

这是一本在国外比较有名的Scheme编程语言的入门教材。本教材适合任何对Scheme编程语言感兴趣的人阅读，尤其是有其他编程语言（特别是动态语言）编程经验，希望快速了解Scheme的不同点并且快速上手写点东西的人。然而希望系统学习Scheme编程的学生仍然是本文的读者之一。

该教程中的大部分内容都能在常见的Scheme入门教材中找到，本教材中比较有特色的是关于 `continuation` 和 `call/cc` 的内容，这也是Scheme的一大特点。第十三章很详细的讲解了 `continuation` 和 `call/cc` ，十四十五章是它们的应用。然而由于本人理解不够深入，这部分（其实整个文章都是）翻译得不堪卒读，所以有问题请对照英文原文理解，也欢迎大家的反馈！

2009年的时候heros翻译了这篇文章的一部分（至第六章未完）。2010年的时候lispor写了一份本教程的读书笔记。2012年的时候又有人试图翻译这篇文章。不过后来貌似没有下文。Scheme的R5RS规范已经在2004年被译成了中文，而这篇实践性比较强的文章却没有完整的中文译版。所以自己翻译了一份（前六章基本是用的hero的版本），本人也是第一次接触Scheme，水平有限，大家多多包涵。

目 录

许可（License）

关 于

献给所有**Scheme**的有缘人

愿智慧仁爱之光永远照耀技术发展的道路

关于本文

该教程中的大部分内容都能在常见的Scheme入门教材中找到，本教材中比较有特色的是关于 `continuation` 和 `call/cc` 的内容，这也是Scheme的一大特点，从这点来说有过一些编程经验（特别是Python和Javascript等动态语言）的程序员会觉得《Teach Yourself Scheme in Fixnum Days》非常适合他们，因为他们只有看一眼马上就明白了Scheme与其他语言相同的地方，因此对某些絮絮叨叨讲语法等基础知识的教程感到厌烦。而这篇文章主要是讲Scheme不同于其他编程语言的地方（不包括语法），以及这种不同是如何应用在Scheme的代码中产生神奇的效果的。当然还有一些命令行和网站CGI的东西，也许某些人希望了解一些。当然你也可以像我一样把它作为学习SICP的入门辅导书。此外，本文还可以作为MzScheme（即现在的Racket语言，之前叫PLT-Scheme）的入门教程。因为本文使用的Scheme实现即是MzScheme。希望更深入学习Racket Lisp但感觉缺乏基础的同志可以看看。

译文缘起及正名

2009年的时候heros翻译了这篇文章的一部分（至第六章未完）。2010年的时候lispor写了一份本教程的读书笔记，翻译了很多内容（而我直到2014年毕设做完也没有看到）。2012年的时候又有人试图翻译这篇文章。不过后来貌似没有下文。Scheme的R5RS规范已经在2004年被译成了中文，而这篇实践性比较强的文章却迟迟没有一个完整的中文译版。所以自己翻译了一份（前六章基本是用的hero的版本），第六章后面又自己翻译了一些，顺便把附录也翻译了。这里要特别感谢我的同学何ufo，虽然他也不是很懂Scheme，不过还是翻译了第七至第十二章，我只是对他的翻译做了一些润色和校验（你发现了其实真正由我翻译的内容不是很多，只是做了一些汇总和润色的工作而已，所以我也不敢以“译者”自居）。本人也是第一次接触Scheme，水平有限，大家多多包涵。后期的任务就是看根据lispor的笔记来校对整个译文。

《Scheme语言简明教程》这个名字已经被用滥了，我见过N篇大同小异的、国内国外的Scheme教程都是这个名字（当然它们都没怎么提 `continuation` 和 `call/cc` ）。所以这样很不利于SEO。。。。不过翻译成《N天学会Scheme》或者《无师自通Scheme语言》又好像有点太俗，而且你发现了作者很聪明的用了一个“Fixnum Day”而不是常见的21天或者3天等等，这让我这个英语水平不怎么样的人很难把意思翻译完整。暂时没想到更好的名称，先就这样吧。

致 谢

首先感谢原作者 Dorai Sitaram 给我们提供这么好的Scheme教程，他一直在维护本文（直到2013年仍有更新）

其次要感谢本文之前的几位翻译者，没有他们的工作我肯定无法把后面的翻译完（估计现在还在前几章打转呢～）

然后还要感谢我的毕设老师，如果没有毕设催着，我肯定没法完成这个工作（虽然大一就打算着看一遍但是大四了也没开始看。。。）

最后感谢王垠学长的博客CSS模板～(写了这么多到底有没有两万汉字啊？？？)

特别的感谢给我的爸爸妈妈，也希望所有看本文的人也都能眞正幸福地生活。

前言

这是一篇Scheme编程语言的介绍。本文的目标是成为一篇快速上手教程。从未接触过Scheme的新手可以在学习更复杂更深入的知识以前通过本文获得一些简明扼要的关于Scheme语言的可实际操作的知识。

本文描述了一种干净利落但实用有效的编写Scheme程序的方法。虽然我们不会按照索引把从A到Z开头的所有函数都介绍一遍，但是我们也不会回避Scheme一些难理解的、凌乱的、非标准、不常用但是却可用或很有用的内容。包括call-with-current-continuation，系统接口和方言的多样性。我们的讨论将围绕我们将解决的问题展开，而不是为了让读者对元语言有什么领悟，因此我并没有按照传统的Scheme教程的思路来撰写本文。本文没有深入的教学方法，没有讲解Scheme语义，没有元循环解释器，也没有讨论Scheme底层的实现，也没有论述Scheme的优点。这并不是说这些东西是不重要的，而是说它们与某些人正在寻找的“快速教程（我是说本文）”无关。

能有多快呢？我不知道一个人是否能在21天学会Scheme[1]，虽然我听有人说精通Scheme的基本内容应该一个下午就够了。Scheme语言的标准——所有精准和复杂的定义都包括进去——只有五十页长。这可能是因为真正对Scheme的大彻大悟（当它到来时），只需一个下午，尽管在那之前不知要花费多少个下午。这就是我的简单介绍。

感谢Matthias Felleisen把Scheme和高阶编程介绍给我，以及Matthew Flatt创造了可靠和优雅的MzScheme实现（本书使用的Scheme即是MzScheme）

==================

> 一个Fixnum是一台机器认为“很小”的一个整数，每个机器对Fixnum都可以有自己的看法

第一章：进入Scheme

经典的第一个程序通常是把一个 `"Hello world!"` 显示在控制台上。用你最喜欢的编辑器，创建一个名为 `hello.scm` 的文件，并在里面输入以下内容：

```
;The first program
(begin
  (display "Hello, World!")
  (newline))
```

第一行是一个注释，当Scheme发现一个分号，就把分号和这一行分号后面的文字都忽略了。

`begin` 语句（原文为form）是Scheme用来包括子语句的方式，这个例子里有两个子语句。第一句调用了 `display` 过程，该过程会输出它的参数（字符串 `"Hello, World!"` ）到控制台（或者叫“标准输出”）后面一句调用了 `newline` 过程，该过程输出一个换行。

想要运行这个程序首先需要启动Scheme，通常只需要在你操作系统的命令行下面输入你的Scheme可执行程序的名字即可。如果你用的是MzScheme，你需要在操作系统提示符后面输入：

```
mzscheme
```

这将调用Scheme listener程序，这个程序读取你的输入，求值，打印结果（如果有的话），然后等待你的下一次输入。由此这通常被称为“读取-求值-打印”的循环。注意这和你操作系统的命令行没有太大区别，操作系统的命令行也读取你的命令，执行，然后等待其他命令。和操作系统一样，Scheme listener有它自己的提示符——通常是 `>` ，但也可能是其他的东西。

在Scheme listener里，加载文件 `hello.scm` 。直接运行下面的语句即可：

```
(load "hello.scm")
```

Scheme现在执行hello.scm文件的内容，输出 `Hello, World!` 接着后面是一个换行符。然后你又得到了命令提示符，可以输入更多命令。

现在由于你有一个很好用的listener，所以你用不着每次把你的程序写到一个文件里然后 `load` 它，有时候，特别是当你想试试某些东西的时候，直接在listener的提示符后面输入表达式然后看结果会更简单。比如，在Scheme的提示符下输入：

```
(begin (display "Hello, World!")
       (newline))
```

会得到：

```
Hello, World!
```

事实上，你可以简单地在提示符后面输入`"Hello, World!"`然后你可以得到作为结果的字符串：

```
"Hello, World!"
```

因为这是listener对`"Hello, World!"`求值的结果。

第二种方式产生的结果除了有双引号以外，两段程序还有一个标志性的区别。第一段(用`begin`开头的)并没有做任何的运算，而显示的结果是`display`和`newline`过程的副作用向标准输出写出来的。第二段程序，`"Hello, World!"`运算得到的结果在这个情况下和这个字符串本身是一致的。

以后，我们会使用标记`=>`来表示运算。就像这样`E => v`表示语句段`E`运算得到结果值为`v`。例如：

```
(begin
  (display "Hello, World!")
  (newline))
=>
```

（没有结果），尽管它有输出`Hello, World!`到标准输出的副作用。

而另一个程序段，

```
"Hello, World!"
=>  "Hello, World!"
```

在上面两种代码情况下，我们运行完后还是停在命令提示符后。如果要退出Scheme，输入`(exit)`，这样会退出Scheme命令行。

Scheme命令行非常便于交互式的测试程序和程序片段。然而这绝不是必须的。你当然可以坚持传统的方式完全在文件中来创建程序，然后用Scheme来执行它们而不使用任何明显的命令行。

在MzScheme中，例如你可以在操作系统的命令行中这样输

```
mzscheme –r hello.scm
```

这样不需要和Scheme命令行打交道就可以产生问候的结果了。 在问候结果产生后，mzscheme将会退回操作系统的命令提示。 这几乎就像是你直接写了`echo Hello , World!`

你甚至可以把`hello.scm`当成是一个系统命令来看待(一个内核脚本或批处理文件)，但具体得等到第十六章来讲解。

第二章 数据类型

数据类型是一组相关的值信息集。各种数据类型互相联系，而且它们通常是具有层次关系。Scheme拥有丰富的数据类型：有一些是简单的类型，还有一些复合类型由其它的类型组合而成。

2.1 简单数据类型

Scheme中的简单数据类型包含 `booleans` (布尔类型) , `number` (数字类型), `characters` (字符类型) 和 `symbols` (标识符类型)。

2.1.1 Booleans

Scheme中的booleans类型用 `#t` 、 `#f` 来分别表示true和false。Scheme拥有一个叫 `boolean?` 的过程，可以用来检测它的参数是否为boolean类型。

```
(boolean? #t)              =>  #t
(boolean? "Hello, World!") =>  #f
```

而 `not` 过程则直接取其参数的相反值做为boolean类型结果。

```
(not #f)              =>  #t
(not #t)              =>  #f
(not "Hello, World!") =>  #f
```

最后一个表达式清晰的显示出了Scheme的一个便捷性：在一个需要boolean类型的上下文中，Scheme会将任何非 `#f` 的值看成true。

2.1.2 Numbers

Scheme的numbers类型可以是 `integers` (整型，例如 `42`)， `rationals` (有理数，例如 `22/7`)， `reals` (实数，例如 `3.14159`)，或 `complex` (复数， `2+3i`)。一个整数是一个有理数，一个有理数是一个实数，一个实数是一个复数，一个复数是一个数字。

Scheme中有可供各种数字进行类型判断的过程：

```
(number? 42)       =>  #t
(number? #t)       =>  #f
(complex? 2+3i)    =>  #t
(real? 2+3i)       =>  #f
(real? 3.1416)     =>  #t
(real? 22/7)       =>  #t
(real? 42)         =>  #t
(rational? 2+3i)   =>  #f
(rational? 3.1416) =>  #t
(rational? 22/7)   =>  #t
(integer? 22/7)    =>  #f
(integer? 42)      =>  #t
```

Scheme的integers(整型)不需要一定是10进制格式。可以通过在数字前加前缀 `#b` 来规定实现2进制。这样 `#b1100` 就是10进制数字12了。实现8进制和16进制格式的前缀分别是 `#o` 和 `#x` 。(decimal前缀 `#d` 是可选项)

我们可以使用通用相等判断过程 `eqv?` 来检测数字的相等性。(`eqv?` 有点类似引用的相等判断ReferenceEquals)

```
(eqv? 42 42)   =>  #t
(eqv? 42 #f)   =>  #f
(eqv? 42 42.0) =>  #f
```

不过，如果你知道参与比较的参数全是数字，选择专门用来进行数字相等判断的 `=` 会更合适些。(`=` 号运算时会根据需要对参数做类型转换，如 `(= 42 "42")` 运算结果是 `#t`)

```
(= 42 42)   =>  #t
(= 42 #f)   -->ERROR!!!
(= 42 42.0) =>  #t
```

其它的数字比较还包括 `<` , `<=` , `>` , `>=`

```
(< 3 2)    =>  #f
(>= 4.5 3) =>  #t
```

`+` , `-` , `*` , `/` , `expt` 等数学运算过程具有我们期待的功能。

```
(+ 1 2 3)    =>  6
(- 5.3 2)    =>  3.3
(- 5 2 1)    =>  2
(* 1 2 3)    =>  6
(/ 6 3)      =>  2
(/ 22 7)     =>  22/7
(expt 2 3)   =>  8
(expt 4 1/2) =>  2.0
```

对于一个参数的情况， `-` 和 `/` 过程会分别得到反数和倒数的结果。

`max` 和 `min` 过程会分别返回提供给它们的参数的最大值和最小值。它们可以支持任何的数字。

```
(max 1 3 4 2 3) =>  4
(min 1 3 4 2 3) =>  1
```

`abs` 过程会返回提供给它参数的绝对值。

```
(abs  3) =>  3
(abs -4) =>  4
```

这些还只是冰山一角。Scheme提供一整套丰富数学和三角运算过程。比如 `atan` , `exp` , 和 `sqrt` 等过程分别返回参数的余切、自然反对数和开方值。

其它更具体的数学运算过程信息请参阅Revised^5 Report on the Algorithmic Language Scheme

---

2.1.3 Characters

Scheme中字符型数据通过在字符前加 `#\` 前缀来表示。像 `#\c` 就表示字符 `c` 。那些非可视字符会有更多的描述名称，例如， `#\newline` , `#\tab` 。空格字符可以写成 `#\ ` ，或者可读性更好一些的 `#\space` 。

字符类型判断过程是 `char?` ：

```
(char? #\c) =>  #t
(char? 1)   =>  #f
(char? #\;) =>  #t
```

需要注意的是数据的分号字符不会引发注释。

字符类型数据有自己的比较判断过程：`char=?`，`char<?`，`char<=?`，`char>?`，`char>=?`

```
(char=? #\a #\a)  =>  #t
(char<? #\a #\b)  =>  #t
(char>=? #\a #\b) =>  #f
```

要实现忽略大小写的比较，得使用`char-ci`过程代替`char`过程：

```
(char-ci=? #\a #\A) =>  #t
(char-ci<? #\a #\B) =>  #t
```

而类型转换过程分别是`char-downcase`和`char-upcase`：

```
(char-downcase #\A) =>  #\a
(char-upcase #\a)   =>  #\A
```

2.1.4 Symbols

前面我们所见到的简单数据类型都是自运算的。也就是如果你在命令提示符后输入了任何这些类型的数据，运算后会返回和你输入内容是一样的结果。

```
#t  =>  #t
42  =>  42
#\c =>  #\c
```

Symbols并没有相同的表现方式。这是因为symbols通常在Scheme程序中被用来当做变量的标识，这样可以运算出变量所承载的值。然而symbols是一种简单数据类型，而且就像characers、numbers以及其它类型数据一样，是Scheme中可以传递的有效值类型。

创建一个单纯的symbol而非变量时，你需要使用`quote`过程：

```
(quote xyz)
=>  xyz
```

因为在Scheme中经常要引用这种类型，我们有一种更简便的方式。表达式`'E`和`(quote E)`在Scheme中是等价的。

Scheme中symbols由一个字符串来命令。在命名时不要和其它类型数据发生冲突，比如characters 、booleans、numbers 或复合类型。
像 `this-is-a-symbol` ， `i18n` ， `<=>` ，和 `$!#*` 都是symbols，而 `16` ， `1+2i` ， `#t` ， `"this-is-a-string"` 和 `'("hello" "world")` 都不是symbols类型数据, `'("hello" "world")` 是一个只包含两个字符串的List。

用来检查symbols类型数据的过程是 `symbol?`

```
(symbol? 'xyz) =>  #t
(symbol? 42)   =>  #f
```

Scheme的symbols类型通常都是不区分大小写的。因此 `Calorie` 和 `calorie` 是等价的

```
(eqv? 'Calorie 'calorie)
=>  #t
```

我们还可以使用 `define` 将symbol 类型的数据 如 `xyz` 当成一个全局的变量来使用：

```
(define xyz 9)
```

这样可以就创建了一个值为9的变量 `xyz` .。 如果现在直接在Scheme命令提示符后输入 `xyz` ，这样会将xyz中的值做为运算结果。

```
xyz
=>  9
```

如果想改变 `xyz` 中的值可以用 `set!` 来实现：

```
(set! xyz #\c)
```

现在 `xyz` 中的值就是字符 `#\c` 了。

```
xyz
=>  #\c
```

2.2 复合数据类型

复合数据类型是以组合的方式通过组合其它数据类型数据来获得。

2.2.1，Strings

字符串类型是由字符组成的序列（不能和symbols混淆，symbols仅是由一组字符来命名的简单类型）。你可以通过将一些字符包上闭合的双引号来得到字符串。Strings是自运算类型。

```
"Hello, World!"
=>  "Hello, World!"
```

还可以通过向 `string` 过程传递一组字符并返回由它们合并成的字符串：

```
(string #\h #\e #\l #\l #\o)
=>  "hello"
```

现在让我们定义一个全局字符串变量 `greeting` 。

```
(define greeting "Hello; Hello!")
```

注意一个字符串数据中的分号不会得到注释。

一个给定字符串数据中的字符可以分别被访问和更改。通过向 `string-ref` 过程传递一个字符串和一个从0开始的索引号，可以返回该字符串指定索引号位置的字符。

```
(string-ref greeting 0)
=>  #\H
```

可以通在一个现有的字符串上追加其它字符串的方式来获得新字符串：

```
(string-append "E "
               "Pluribus "
               "Unum")
=>  "E Pluribus Unum"
```

你可以定义一个指定长度的字符串，然后用期望的字符来填充它。

```
(define a-3-char-long-string (make-string 3))
```

检测一个值是否是字符串类型的过程是 `string?` 。

通过调用 `string` ， `make-string` 和 `string-append` 获得的字符串结果都是可修改的。而过程 `string-set!` 就可以替换字符串指定索引处的字符。

```
(define hello (string #\H #\e #\l #\l #\o))
hello
=>  "Hello"

(string-set! hello 1 #\a)
hello
=>  "Hallo"
```

2.2.2 Vectors (向量)

Vectors是像strings一样的序列，但它们的元素可以是任何类型，而不仅仅是字符，当然元素也可以是Vetors类型，这是一种生成多维向量的好方式。

这使用五个整数创建了一个vector：

```
(vector 0 1 2 3 4)
=>  #(0 1 2 3 4)
```

注意Scheme表现一个向量值的方式：在用一对小括号包括起来的向量元素前面加了一个 `#` 字符。

和 `make-string` 过程类似，过程 `make-vectors` 可以构建一个指定长度的向量：

```
(define v (make-vector 5))
```

而过程 `vector-ref` 和 `vector-set!` 分别可以访问和修改向量元素。

检测值是否是一个向量的过程是 `vector?` 。

2.2.3 Dotted pairs(点对) 和 lists(列表)

点对是将两个任意数值组合成有序数偶的复合类型。点对的第一个数值被称作car，第二值被称作cdr，而将两个值组合成点值对的过程是cons。

```
(cons 1 #t)
=>  (1 . #t)
```

点对不能自运算，因此直接以值的方式来定义它们（即不通过调用 `cons` 来创建），必须显式的使用引号：

```
'(1 . #t) =>  (1 . #t)

(1 . #t)  -->ERROR!!!
```

访问点值对值的过程分别是`car`(`car`访问点值对的第一个元素)和`cdr`(`cdr`访问点值对的非一个元素):

```
(define x (cons 1 #t))

(car x)
=>  1

(cdr x)
=>  #t
```

点对的元素可以通过修改器过程`set-car!`和`set-cdr!`来进行修改:

```
(set-car! x 2)

(set-cdr! x #f)

x
=>  (2 . #f)
```

点对也可以包含其它的点对。

```
(define y (cons (cons 1 2) 3))

y
=>  ((1 . 2) . 3)
```

这个点对的`car`运算结果`car`运算结果是`1`,而`car`运算结果的`cdr`运算结果是`2`。即:

```
(car (car y))
=>  1

(cdr (car y))
=>  2
```

Scheme提供了可以简化`car`和`cdr`组合起来连续访问操作的简化过程。像`caar`表示"`car`运算结果的`car`运算结果", `cdar`表示"`car`运算结果的`cdr`运算结果",等等。

```
(caar y)
=>  1

(cdar y)
=>  2
```

像c...r这样风格的简写最多只支持四级连续操作。像 `cadr` ， `cdadr` ，和 `cdaddr` 都是存在的。而 `cdadadr` 这样的就不对了。

当第二个元素是一个嵌套的点对时，Scheme使用一种特殊的标记来表示表达式的结果：

```
(cons 1 (cons 2 (cons 3 (cons 4 5))))
=>  (1 2 3 4 . 5)
```

即， `(1 2 3 4 . 5)` 是对 `(1 . (2 . (3 . (4 . 5))))` 的一种简化。这个表达式的最后一个 `cdr` 运算结果是5。

如果嵌套点值对最后一个 `cdr` 运算结果是一个空列表对象，Scheme提供了一种更进一步的用表达式 `'()` 来表示的简化方式。 空列表没有被考虑做为可以自运算的值，所以为程序提供一个空列表值时必须用单引号方式来创建：

```
'() =>  ()
```

诸如像 `(1 . (2 . (3 . (4 . ()))))` 这样形式的点值对被简化成 `(1 2 3 4)` 。像这样第二元素都是一个点值对特殊形式的嵌套点值对就称作列表list。这是一个四个元素长度的列表。可以像这样来创建：

```
(cons 1 (cons 2 (cons 3 (cons 4 '()))))
```

但Scheme提供了一个list过程可以更方便的创建列表。List可以将任意个数的参数变成列表返回：

```
(list 1 2 3 4)
=>  (1 2 3 4)
```

实际上，如果我们知道列表所包含的所有元素，我们还可以用 `quote` 来定义一个列表：

```
'(1 2 3 4)
=>  (1 2 3 4)
```

列表的元素可以通过指定索引号来访问。

```
(define y (list 1 2 3 4))

(list-ref y 0) =>  1
(list-ref y 3) =>  4

(list-tail y 1) =>  (2 3 4)
(list-tail y 3) =>  (4)
```

`list-tail` 返回了给定索引号后的所有元素。

`pair?` ， `list?` 和 `null?` 判断过程可以分别用来检查它们的参数是不是一个点对，列表或空列表。

```
(pair? '(1 . 2)) =>  #t
(pair? '(1 2))   =>  #t
(pair? '())      =>  #f
(list? '())      =>  #t
(null? '())      =>  #t
(list? '(1 2))   =>  #t
(list? '(1 . 2)) =>  #f
(null? '(1 2))   =>  #f
(null? '(1 . 2)) =>  #f
```

2.2.1 数据类型转换

Scheme提供了许多可以进行数据类型转换的过程。我们已经知道可以通过 `char-downcase` 和 `char-upcase` 过程来进字符大小写的转换。字符还可以通过使用 `char->integer` 来转换成整型，同样的整型也可以通过 `integer->char` 被转换成字符。(字符转换成整型得到的结果通常是这个字符的ascii码值。)

```
(char->integer #\d) =>  100
(integer->char 50)  =>  #\2
```

字符串可以被转换成等价的字符列表。

```
(string->list "hello") =>  (#\h #\e #\l #\l #\o)
```

其它的转换过程也都是一样的风格 `list->string` ， `vector->list` 和 `list->vector` 。

数字可以转换成字符串： `(number->string 16) => "16"`

字符串也可以转换成数字。如果字符串不能转换成数字，则会返回 `#f` 。

```
(string->number "16")
=>  16

(string->number "Am I a not number?")
=>  #f
```

`string->number` 第二个参数是可选参数，指示以几进制来转换。

```
(string->number "16" 8) =>  14
```

八进制的数字 `16` 等于 `14` 。

Symbols也可以转换为字符串，反之亦然：

```
(symbol->string 'symbol)
=>  "symbol"

(string->symbol "string")
=>  string
```

2.3 其它数据类型

Scheme还包含了一些其它数据类型。一个是 *procedure* (过程)。我们已经见过了许多过程了，例如， `display` ， `+` ， `cons` 等。实际上，它们是一些承载了过程值的变量，过程本身内部的数值和字符并不可见：

```
cons
=>  <procedure>
```

迄今为止我们所见过的这些过程都属于原始过程（系统过程），由一些全局变量来承载它们。用户还可以添加自定义的过程。

还有另外种数据类型是port端口。一个端口是为输入输出提供执行的通道。端口通常会和文件和控制台操作相关联。

在我们的 `"Hello，World！"` 程序中，我们使用 `display` 过程向控制台输出了一个字符串。 `display` 可以接受两个参数，第一个参数值是将输出的值，另一个值则表示了即将承载显示结果的输出port(端口)。

在我们的程序中， `display` 的第二参数是隐式参数。这时候 `display` 会采用标准输出端口作为它的默认输出端口。我们可以通过调用 `current-output-port` 过程来取得当前的标准输出端口。我们可以更清楚的写出:

```
(display "Hello, World!" (current-output-port))
```

2.4 S-expressions（S表达式）

所有这些已经被讨论过的数据类型可以被统一成一种通用的叫作s-expression(符号表达式或s-表达式)的数据类型(s代表符号)。像 `42` ，`#\c` ，`(1 . 2)` ，`#(a b c)` ，`"Hello"` ，`(quote xyz)` ，`(string->number "16")` ，和 `(begin (display "Hello, World!") (newline))` 都是s-表达式。

第三章 Forms代码结构

读者们会发现迄今为止我们提供的Scheme示例程序也都是s-表达式。这对所有的Scheme程序来说都适用：程序是数据。

因此，字符数据 `#\c` 也是一个程序，或一个代码结构。我们将使用更通用的说法代码结构而不是程序，这样我们也可以处理程序片段。

Scheme计算代码结构 `#\c` 得到结果 `#\c` ，因为 `#\c` 可以自运算。但不是所有的s-表达式都可以自运算。比如symbol 表达式 `xyz` 运算得到的结果是 `xyz` 这个变量所承载的值；list 表达式 `(string->number "16")` 运算的结果是数字 `16` 。(注：之前学过的list类型的数据都是类似 `(1 2 3 4 5)` 这样，所以称 `(string->number "16")` 为列表s-表达式)

也不是所有的s-表达式都是有效的程序。如果你直接输入点值对 `(1 . 2)` ，你将会得到一个错误。

Scheme运行一个列表形式的代码结构时，首先要检测列表第一个元素，或列表头。如果这个列表头是一个过程，则代码结构的其余部分则被当成将传递给这个过程的参数集，而这个过程将接收这些参数并运算。

如果这个代码结构的列表头是一个特殊的代码结构，则将会采用一种特殊的方式来运行。我们已经碰到过的特殊的代码结构有 `begin` ， `define` 和 `set!` 。

`begin` 可以让它的子结构可以有序的运算，而最后一个子结构的结果将成为整个代码结构的运行结果。 `define` 会声明并会初始化一个变量。 `set!` 可以给已经存在的变量重新赋值。

3.1 Procedures(过程)

我们已经见过了许多系统过程，比如， `cons` ， `string->list` 等。用户可以使用代码结构 `lambda` 来创建自定义的过程。例如，下面定义了一个过程可以在它的参数上加上2：

```
(lambda (x) (+ x 2))
```

第一个子结构， `(x)` ，是参数列表。其余的子结构则构成了这个过程执行体。这个过程可以像系统过程一样，通过传递一个参数完成调用：

```
((lambda (x) (+ x 2)) 5)
=>  7
```

如果我们希望能够多次调用这个相同的过程，我们可以每次使用 `lambda` 重新创建一个复制品，但我们有更好的方式。我们可以使用一个变量来承载这个过程：

```
(define add2
  (lambda (x) (+ x 2)))
```

只要需要，我们就可以反复使用 `add2` 为参数加上2：

```
(add2 4) =>  6
(add2 9) =>  11
```

译者注:定义过程还可以有另一种简单的方式，直接用define而不使用lambda来创建：

```
(define (add2 x)
       (+ x 2))
```

3.1.1 过程的参数

`lambda` 过程的参数由它的第一个子结构（紧跟着 `lambda` 标记的那个结构）来定义。 `add2` 是一个单参数或一元过程，所以它的参数列表是只有一个元素的列表 `(x)` 。标记 `x` 作为一个承载过程参数的变量而存在。在过程体中出现的所有 `x` 都是指代这个过程的参数。对这个过程体来说x是一个局部变量。

我们可以为两个参数的过程提供两个元素的列表做参数，通常都是为n个参数的过程提供n个元素的列表。下面是一个可以计算矩形面积的双参数过程。它的两个参数分别是矩形的长和宽。

```
(define area
  (lambda (length breadth)
    (* length breadth)))
```

我们看到 `area` 将它的参数进行相乘，系统过程 `*` 也可以实现相乘。我们可以简单的这样做：

```
(define area *)
```

3.1.2 可变数量的参数（不定长参数）

有一些过程可以在不同的时候传给它不同个数的参数来完成调用。为了实现这样的过程， `lambda` 表达式列表形式的参数要被替换成单个的符号。这个符号会像一个变量一样来承载过程调用时接收到的参数列表。

通常， `lambda` 的参数列表可以是一个列表构结 `(x …)` ，一个符号，或者 `(x … . z)` 这样的一个点对结构。

当参数是一个点对结构时，在点之前的所有变量将一一对应过程调用时的前几个参数，点之后的那个变量会将剩余的参数值作为一个列表来承载。

3.2 apply过程

`apply` 过程允许我们直接传递一个装有参数的list 给一个过程来完成对这个过程的批量操作。

```
(define x '(1 2 3))

(apply + x)
=>  6
```

通常， `apply` 需要传递一个过程给它，后面紧接着是不定长参数，但最后一个参数值一定要是list。它会根据最后一个参数和中间其它的参数来构建参数列表。然后返回根据这个参数列表来调用过程得到的结果。例如:

```
(apply + 1 2 3 x)
=>  12
```

3.3 顺序执行

我们使用 `begin` 这个特殊的结构来对一组需要有序执行的子结构来进行打包。许多Scheme的代码结构都隐含了 `begin` 。例如，我们定义一个三个参数的过程来输出它们，并用空格间格。一种正确的定义是：

```
(define display3
  (lambda (arg1 arg2 arg3)
    (begin
      (display arg1)
      (display " ")
      (display arg2)
      (display " ")
      (display arg3)
      (newline))))
```

在Scheme中，lambda的语句体都是隐式的 `begin` 代码结构。因此， `display3` 语句体中的begin不是必须的，不写时也不会有什么影响。

`display3` 更简化的写法是：

```
(define display3
  (lambda (arg1 arg2 arg3)
    (display arg1)
    (display " ")
    (display arg2)
    (display " ")
    (display arg3)
    (newline)))
```

第四章，条件语句

和其它的编程语句一样，Scheme 也包含条件语句。

最基本的结构就是if：

```
(if 测试条件
    then-分支
    else-分支)
```

如果测试条件运算的结果是真(即，非 `#f` 的任何其它值)， `then` 分支将会被运行(即满足条件时的运行分支)。否则， `else` 分支会被运行。 `else` 分支是可选的。

```
(define p 80)

(if (> p 70)
    'safe
    'unsafe)
=>  safe

(if (< p 90)
    'low-pressure) ;no ``else'' branch
=>  low-pressure
```

为了方便，Scheme还提供了一些其它的条件结构语句。它们可以被定义成宏来扩充if表达式。

4.1 when 和 unless

当我们只需要一个基本条件语句分支时（”then”分支或”else”分支），使用when 和unless会更方便。(这里的示例已经更换，原示例)

```
(define a 10)
(define b 20)
(when (< a b)
       (display “a是”)
       (display a)
       (display “b是”)
       (display b)
       (display “a大于b” ) )
```

先判断a是否小于b，这个条件成立时会输出5条信息。

使用if实现相同的程序会是这样：

```
(define a 10)
(define b 20)
(if (< a b)
       (begin
              (display “a是”)
              (display a)
              (display “b是”)
              (display b)
              (display “a大于b” ) ))
```

注意 `when` 的分支是一个隐式的 `begin` 语句结构， 而如果 `if` 的分支有多个代码结构时，需要一个显式的 `begin` 代码结构。

同样的功能还可以像下面这样用 `unless` 来写(`unless` 和 `when` 的意思正好相反)：

```
(define a 10)
(define b 20)
(unless (>= a b)
       (display “a是”)
       (display a)
       (display “b是”)
       (display b)
       (display “a大于b” ) )
```

并不是所有的Scheme环境都提供 `when` 和 `unless` 。 如果你的Scheme中没有，你可以用宏来自定义出 `when` 和 `unless` (宏，见第8章)。

4.2 cond

`cond` 结构在表示多重 `if` 表达式时很方便， 多重 `if` 结构除了最后一个 `else` 分支以外的其余分支都会包含一个新的 `if` 条件。因此，

```
(if (char<? c #\c) -1
    (if (char=? c #\c) 0
        1))
```

这样的结构都可以使用 `cond` 来这样写：

```
(cond ((char<? c #\c) -1)
      ((char=? c #\c) 0)
      (else 1))
```

`cond` 就是这样的一种多分支条件结构。每个从句都包含一个判断条件和一个相关的操作。第一个判断成立的从句将会引发它相关的操作执行。如果任何一个分支的条件判断都不成立则最后一个 `else` 分支将会执行(`else` 分支语句是可选的)。

cond的分支操作都是 `begin` 结构。

4.3 case

当 `cond` 结构的每个测试条件是一个测试条件的分支条件时，可以缩减为一个 `case` 表达式。

```
(define c #\c)
(case c
  ((#\a) 1)
  ((#\b) 2)
  ((#\c) 3)
  (else 4))
=>  3
```

分支头值是 `#\c` 的分支将被执行。

4.4 and 和 or

Scheme提供了对boolean值进行逻辑与 `and` 和逻辑或 `or` 运算的结构。(我们已经见过了布尔类型的求反运算not过程。)

当所有子结构的值都是真时， `and` 的返回值是真，实际上， `and` 的运行结果是最后一个子结构的值。如果任何一个子结构的值都是假，则返回 `#f` 。

```
(and 1 2)  =>  2
(and #f 1) =>  #f
```

而 `or` 会返回它第一个为值为真的子结构的结果。如果所有的子结构的值都为假， `or` 则返回 `#f` 。

```
(or 1 2)  =>  1
(or #f 1) =>  1
```

`and` 和 `or` 都是从左向右运算。当某个子结构可以决定最终结果时， `and` 和 `or` 会忽略剩余的子结构，即它们是“短路”的。

```
(and 1 #f expression-guaranteed-to-cause-error)
=>  #f

(or 1 #f expression-guaranteed-to-cause-error)
=>  1
```

第五章，词法变量

Scheme的变量有一定的词法作用域，即它们在程序代码中只对特定范围的代码结构可见。迄今为止我们所见过的全局变量也没有例外的：它们的作用域是整个程序，这也是一种特定的作用范围。

我们也碰见过一些示例包含局部变量。它们都是lambda过程的参数，当过程被调用时这些变量会被赋值，而它们的作用域仅限于在过程的内部。例如：

```
(define x 9)
(define add2 (lambda (x) (+ x 2)))


x  =>  9

(add2 3) =>  5
(add2 x) =>  11

x  =>  9
```

这里有一个全局变量 `x` ，还有一个局部变量 `x` ，就是在过程 `add2` 中那个字母 `x` 。全局变量 `x` 的值一直是9。第一次调用 `add2` 过程时，局部的 `x` 会被赋值为3，而第二次调用 `add2` 时，局部变量 `x` 的会被赋值为全局变量 `x` 的值，即 `9` 。

当过程的调用结束时，全部变量 `x` 仍然是9。

而 `set!` 代码结构可修改变量的赋值。

```
(set! x 20)
```

上面代码将全局变量 `x` 的值9修改为20，因为对于 `set!` 全局变量是可见的。如果 `set!` 是在 `add2` 过程体内被调用，那修改的就是局部变量 `x` ：

```
(define add2
  (lambda (x)
    (set! x (+ x 2))
    x))
```

这里 `set!` 在局部变量 `x` 上加上2，并且会返回局部变量x的新值。(从结果来看，我们无法区分这个过程和先前的 `add2` 过程)。

我们可以像先前一样使用全局的 `x` 做参数值来调用 `add2` ：

```
(add2 x) =>  22
```

(记住全局变量x的值现在是20，而不是9!)

`add2` 过程内的 `set!` 调用仅会影响局部变量x。尽管局部变量x被赋了全局变量x的值，但后者不会因为 `set!` 为局部变量 `x` 赋值而受影响。

```
x =>  20
```

注意我们做这些讨论是因为我们为局部变量和全局变量使用了同样的标识 `x` 。在某些代码中，这个叫 `x` 的标识符指的是语法闭包中的局部 `x` 变量，这会暂时隐藏闭包外或全局变量 `x` 的值。例如，

```
(define counter 0)

(define bump-counter
  (lambda ()
    (set! counter (+ counter 1))
    counter))
```

`bump-counter` 是一个没有参数的过程(没有参数的过程也称作 `thunk`). 它没有引入局部变量和参数，这样就不会隐藏任何值。在每次调用时，它会修改全局变量 `counter` 的值，让它增加1，然后返回它当前的值。下面是一些 `bump-counter` 的成功调用示例:

```
(bump-counter) =>  1
(bump-counter) =>  2
(bump-counter) =>  3
```

5.1 let 和 let*

并不是一定要显式的创建过程才可以创建局部变量。有个特殊的代码结构let可以创建一列局部变量以便在其结构体中使用:

```
(let ((x 1)
      (y 2)
      (z 3))
  (list x y z))
=>  (1 2 3)
```

和 `lambda` 一样，在 `let` 结构体中，局部变量 `x` （赋值为1）会暂时隐藏全局变量 `x` （赋值为20）。

局部变量 `x` 、 `y` 、 `z` 分别被赋值为1、2、3，这个初始化的过程并不作为 `let` 过程结构体的一部分。因此，在初始化时对 `x` 的引用都指向了全局变量 `x` ，而不是局部变量 `x` 。

```
(let ((x 1)
      (y x))
  (+ x y))
=>  21
```

上面代码中，因为局部变量 `x` 被赋值为1，而 `y` 被赋上了值为20的全局变量 `x` 。

有时候，用 `let` 依次的创建局变量非常的方便，如果在初始化区域中可以用先创建的变量来为后创建的变量赋值也会非常方便。 `let*` 结构就可以这样做：

```
(let* ((x 1)
       (y x))
  (+ x y))
=>  2
```

在初始化y变量时的x，指的是前面刚创建好的变量x。 这个例子完全等价于下面这个 `let` 嵌套的程序，更深了说，实际上就是 `let` 嵌套的缩写。

```
(let ((x 1))
  (let ((y x))
    (+ x y)))
=>  2
```

我们也可以把一个过程做为值赋给变量：

```
(let ((cons (lambda (x y) (+ x y))))
  (cons 1 2))
=>  3
```

在这个 `let` 构结体中，变量 `cons` 将它的参数进行相加。而在 `let` 结构的外面， `cons` 还是用来创建点对。

5.2 fluid-let

一个词法变量如果没有被隐藏，在它的作用域内一直都为可见状态。有时候，我们有必要将一个词法变量临时的设置为一个固定的值。为此我们可使用 `fluid-let` 结构(`fluid-let` 是一个非标准的特殊结构。可参见8.3，在Scheme中定义fluid-let)。

```
(fluid-let ((counter 99))
  (display (bump-counter)) (newline)
  (display (bump-counter)) (newline)
  (display (bump-counter)) (newline))
```

这和 `let` 看起来非常相像，但并不是暂时的隐藏了全局变量counter的值，而是在 `fluid-let` 执行体中临时的将全局变量 `counter` 的值设置为了99直到执行体结束。因此执行体中的三句 `display` 产生了结果

```
100
101
102
```

当 `fluid-let` 表达式计算结束后，全局变量 `counter` 会恢复成之前的的值。

```
counter =>  3
```

注意 `fluid-let` 和 `let` 的效果完全不同。 `fluid-let` 不会和 `let` 一样产生一个新的变量。它会修改已经存的变量的值绑定，当 `fluid-let` 结束时这个修改也会结束。

为了清楚的说明这一些，可以思考这个根据前一个示例用 `let` 替换 `fluid-let` 后的程序。这次的输出是

```
4
5
6
```

即，初始值为3的全局变量 `counter` ，被每一次 `bump-counter` 的调用更新。而新创建的初始值为99的词法变量 `counter` 并没有影响到 `bump-counter` 的执行，因为尽管 `bump-counter` 是在局部变量 `counter` 的作用域内被调用的，但 `bump-counter` 的结构体并不在这个作用域内。所以 `bump-counter` 中的 `counter` 仍然指的是全局变量 `counter` ，最后的值为6。

```
counter =>  6
```

第六章，递归

一个过程体中可以包含对其它过程的调用，特别的是也可以调用自己。

```
(define factorial
 (lambda (n)
   (if (= n 0) 1
       (* n (factorial (- n 1))))))
```

这个递归过程用来计算一个数的阶乘。如果这个数是0，则结果为1。对于任何其它的值n，这个过程会调用其自身来完成n-1阶乘的计算，然后将这个子结果乘上n并返回最终产生的结果。

互递归过程也是可以的。下面判断奇偶数的过程相互进行了调用。

```
(define is-even?
 (lambda (n)
   (if (= n 0) #t
       (is-odd? (- n 1)))))

(define is-odd?
 (lambda (n)
   (if (= n 0) #f
       (is-even? (- n 1)))))
```

这里提供的两个过程的定义仅作为简单的互递归示例。Scheme已经提供了简单的判断过程 `even?` 和 `odd?` 。

6.1 letrec

如果希望将上面的过程定义为局部的，我们会尝试使用let结构：

```
(let ((local-even? (lambda (n)
                     (if (= n 0) #t
                         (local-odd? (- n 1)))))
      (local-odd? (lambda (n)
                    (if (= n 0) #f
                        (local-even? (- n 1))))))
 (list (local-even? 23) (local-odd? 23)))
```

但这并不能成功，因为在初始化值过程中出现的 `local-even?` 和 `local-odd?` 指向的并不是这两个过程本身。

把 `let` 换成 `let*` 同样也不能奏效，因为这时虽然 `local-odd?` 中出现的 `local-even?` 指向的是前面刚创建好的局部的过程，但 `local-even?` 中的 `local-odd?` 还是指向了别处。

为解决这个问题，Scheme提供了 `letrec` 结构。

```
(letrec ((local-even? (lambda (n)
                        (if (= n 0) #t
                            (local-odd? (- n 1)))))
         (local-odd? (lambda (n)
                       (if (= n 0) #f
                           (local-even? (- n 1))))))
 (list (local-even? 23) (local-odd? 23)))
```

用 `letrec` 创建的词法变量不仅可以在 `letrec` 执行体中可见而且在初始化中也可见。 `letrec` 是专门为局部的递归和互递归过程而设置的。(这里也可以使用 `define` 来创建两个子结构的方式来实现局部递归)

6.2 命名let

使用 `letrec` 定义递归过程可以实现循环。如果我们想显示10到1的降数列，可以这样写：

```
(letrec ((countdown (lambda (i)
                      (if (= i 0) 'liftoff
                          (begin
                            (display i)
                            (newline)
                            (countdown (- i 1)))))))
 (countdown 10))
```

这会在控制台上输出10到1，并会返回结果 `liftoff` 。

Scheme允许使用一种叫"命名let"的 `let` 变体来更简洁的写出这样的循环:

```
(let countdown ((i 10))
 (if (= i 0) 'liftoff
      (begin
        (display i)
        (newline)
        (countdown (- i 1)))))
```

注意在 `let` 的后面立即声明了一个变量用来表示这个循环。这个程序和先前用 `letrec` 写的程序是等价的。你可以将"命名let"看成一个对 `letrec` 结构进行扩展的宏。

6.3 迭代

上面定义的 `countdown` 函数事实上是一个递归的过程。Scheme只有通过递归才能定义循环，不存在特殊的循环或迭代结构。

尽管如此，上述定义的循环是一个“眞”循环，与其他语言实现它们的循环的方法完全相同。也就是说,Scheme十分注意确保上面使用过的递归类型不会产生过程调用/返回开销。

Scheme通过一种消除尾部调用（tail-call elimination）的过程完成这个功能。如果你注意观察 `countdown` 的步骤，你会注意到当递归调用出现在 `countdown` 主体内时，就变成了“尾部调用”，或者说是最后完成的事情—— `countdown` 的每次调用要么不调用它自身，要么当它调用自身时把这个动作留在最后。对于一个Scheme语言的实现来说（解释器），这会使递归不同于迭代。因此，尽管用递归去写循环吧，这是安全的。

这是又一个有用的尾递归程序的例子：

```
(define list-position
  (lambda (o l)
    (let loop ((i 0) (l l))
      (if (null? l) #f
          (if (eqv? (car l) o) i
              (loop (+ i 1) (cdr l)))))))
```

`list-position` 发现了 `o` 对象在列表 `l` 中第一次出现的索引。如果在列表中没有发现对象，过程将会返回 `#f` 。

这又是一个尾部递归过程，它将自身的参数列表就地反转，也就是使现有的列表内容产生变异，而没有分配一个新的列表：

```
(define reverse!
  (lambda (s)
    (let loop ((s s) (r '()))
      (if (null? s) r
      (let ((d (cdr s)))
            (set-cdr! s r)
        (loop d s))))))
```

`reverse!` 是一个十分有用的过程，它在很多Scheme方言中都能使用，例如MzScheme和Guile）

更多地递归例子（包括迭代）参见附录C。

6.4 用自定义过程映射整个列表

有一种特殊类型的迭代，对列表中每个元素，它都会重复相同的动作。Scheme为这种情况提供了两种程序：`map` 和 `for-each` 。

`map` 程序为给定列表中的每个元素提供了一种既定程序，并返回一个结果的列表。例如：

```
(map add2 '(1 2 3))
=>  (3 4 5)
```

`for-each` 程序也为列表中的每个元素提供了一个程序，但返回值为空。这个程序纯粹是产生的副作用。例如：

```
(for-each display
  (list "one " "two " "buckle my shoe"))
```

这个程序在控制台上有显示字符串（在它们出现的顺序上）的副作用。

这个由 `map` 和 `for-each` 用在列表上的程序并不一定是单参数程序。举例来说，假设一个n参数的程序， `map` 会接受n个列表，每个列表都是由一个参数所组成的集合，而 `map` 会从每个列表中取相应元素提供给程序。例如：

```
(map cons '(1 2 3) '(10 20 30))
=>  ((1 . 10) (2 . 20) (3 . 30))

(map + '(1 2 3) '(10 20 30))
=>  (11 22 33)
```

第七章 输入输出

Scheme的输入/输出程序可以使你从输入端口读取或者将写入到输出端口。端口可以关联到控制台，文件和字符串。

7.1 读取

Scheme的读取程序带有一个可选的输入端口参数。如果端口没有特别指定，则假设为当前端口（一般是控制台）。

读取的内容可以是一个字符，一行数据或是S表达式。当每次执行读取时，端口的状态就会改变，因此下一次就会读取当前已读取内容后面的内容。如果没有更多的内容可读，读取程序将返回一个特殊的数据——文件结束符或EOF对象。这个对象只能用 `eof-object?` 函数来判断。

`read-char` 程序会从端口读取下一个字符。 `read-line` 程序会读取下一行数据，并返回一个字符串（不包括最后的换行符）， `read` 程序则会读取下一个S表达式。

7.2 写入

Scheme的写入程序接受一个要被写入的对象和一个可选的输出端口参数。如果未指定端口，则假设为当前端口（一般为控制台）。

写入的对象可以是字符或是S表达式。

`write-char` 程序可以向输出端口写入一个给定的字符（不包括 `#\` ）。 `write` 和 `display` 程序都可以向端口写入一个给定的S表达式，唯一的区别是： `write` 程序会使用机器可读型的格式而 `display` 程序却不用。例如， `write` 用双引号表示字符串，用 `#\` 句法表示字符，但 `display` 却不这么做。

`newline` 程序会在输出端口输出一个换行符。

7.3 文件端口

如果端口是标准的输入和输出端口，Scheme的I/O程序就不需要端口参数。但是，如果你明确需要这些端口，则 `current-input-port` 和 `current-output-port` 这些零参数程序会提供这个功能，例如：

```
(display 9)
(display 9 (current-output-port))
```

拥有相同的效果。

一个端口通过打开文件和这个文件关联在一起。 `open-input-file` 程序会接受一个文件名作为参数并返回一个和这个文件关联的新的输入端口。 `open-output-file` 程序会接受一个文件名作为参数并返回一个和这个文件关联的新的输出端口。如果打开一个不存在的输入文件，或者打开一个已经存在的输出文件，程序都会出错。

当你已经在一个端口执行完输入或输出后，你需要使用 `close-input-port` 或 `close-output-port` 程序将它关闭。

在下述例子中，假如文件 `hello.txt` 文件只包含一个单词 `hello` 。

```
(define i (open-input-file "hello.txt"))

(read-char i)
=>  #\h

(define j (read i))

j
=>  ello
```

假如文件 `greeting.txt` 在下述程序运行前不存在：

```
(define o (open-output-file "greeting.txt"))

(display "hello" o)
(write-char #\space o)
(display 'world o)
(newline o)

(close-output-port o)
```

现在 `Greeting.txt` 文件将会包含这样一行：

```
hello world
```

7.3.1 文件端口的自动打开和关闭

Scheme提供了 `call-with-input-file` 和 `call-with-output-file` 过程，这些过程会照顾好打开的端口并在你使用完后将端口关闭。

`call-with-input-file` 程序接受一个文件名参数和一个过程。这个过程被应用在一个已打开的文件输入端口。当程序结束时，它的结果会在保证端口关闭后返回。

```
(call-with-input-file "hello.txt"
  (lambda (i)
    (let* ((a (read-char i))
           (b (read-char i))
           (c (read-char i)))
      (list a b c))))
=>  (#\h #\e #\l)
```

`call-with-output-file` 程序会对输出文件提供类似的服务。

7.4 字符串端口

一般来说将字符串与端口相关联是很方便的。因此，`open-input-string` 程序将一个给定的字符串和一个端口关联起来。读取这个端口的程序将读出下述字符串：

```
(define i (open-input-string "hello world"))

(read-char i)
=>  #\h

(read i)
=>  ello

(read i)
=>  world
```

`open-output-string` 创建了一个输出端口，最终可以用于创建一个字符串：

```
(define o (open-output-string))

(write 'hello o)
(write-char #\, o)
(display " " o)
(display "world" o)
```

现在你可以使用 `get-output-string` 程序得到保留在字符串端口 `o` 中的字符串：

```
(get-output-string o)
=>  "hello, world"
```

字符串端口不需要显式地去关闭。

7.5 加载文件

我们已将看到`load`程序可以加载包含`Scheme`代码的文件。`load`一个文件意味着按顺序求值文件中每一个Scheme表达式。`load`中的路径参数是相对当前Scheme工作目录计算的，该工作目录一般是调用Scheme可执行文件时的目录。

一个文件可以加载其他的文件，这在包含许多文件的大项目中十分有用。但是，除非使用绝对路径，否则`load`参数中的文件位置将依赖于执行Scheme的当前目录。而提供绝对路径名并不是很方便，因为我们更愿意把项目文件作为一个单元（保留它们的相对路径名）在很多不同机器中运行。

Mzscheme提供了`load-relative`程序，可以很好的解决这个问题。`load-relative`，和`load`相似，带有一个路径名参数。当在`foo.scm`文件中出现`load-relative`调用时，它的参数的路径将根据文件`foo.scm`所在目录的路径来计算。特别注意的是，这个路径名和执行Scheme的当前目录无关，因此也就可以方便地进行多文件程序的开发。

第八章 宏

用户可以通过定义宏来创建属于自己的 `special form` 。宏是一个具有与它相关联的转换器程序的标记。当Scheme遇到一个宏表达式，即以macro—作为开头的列表时，它会将宏的转换器应用于宏表达式中的子列表，而且会对最后的转换结果进行求值。

理想情况下，“宏”指代从一种代码文本到另一种代码文本的纯文本变换。这种变换对于缩写那些复杂的但经常出现的文本模式十分有用。

宏通过 `define-macro` 来定义（见附录A.3）。例如，如果你的Scheme缺少条件表达式when，你就可以以下述宏定义when：

```
(define-macro when
  (lambda (test . branch)
    (list 'if test
      (cons 'begin branch))))
```

这样定义的when转换器能够把一个when表达式转换为等价的if表达式。用这个宏，下面的when表达式

```
(when (< (pressure tube) 60)
   (open-valve tube)
   (attach floor-pump tube)
   (depress floor-pump 5)
   (detach floor-pump tube)
   (close-valve tube))
```

将会被转换为另一个表达式，把when转换器应用到when表达式的子 `form` ：

```
(apply
  (lambda (test . branch)
    (list 'if test
      (cons 'begin branch)))
  '((< (pressure tube) 60)
      (open-valve tube)
      (attach floor-pump tube)
      (depress floor-pump 5)
      (detach floor-pump tube)
      (close-valve tube)))
```

这个转换产生了一个列表：

```
(if (< (pressure tube) 60)
    (begin
      (open-valve tube)
      (attach floor-pump tube)
      (depress floor-pump 5)
      (detach floor-pump tube)
      (close-valve tube)))
```

Scheme将会对这个表达式进行求值，就像它对其他表达式所做的一样。

再来看另一个例子，这有一个`unless`（`when`的另一种形式）的宏定义：

```
(define-macro unless
  (lambda (test . branch)
    (list 'if
          (list 'not test)
          (cons 'begin branch))))
```

另外，我们可以调用`when`放进`unless`定义中：

```
(define-macro unless
  (lambda (test . branch)
    (cons 'when
          (cons (list 'not test) branch))))
```

宏表达式可以引用其他的宏。

8.1 指定一个扩展为模板

宏转换器一般接受一些S表达式作为参数，同时产生可以被作为`form`使用的S表达式。通常情况下输出是一个列表。在我们的when例子中，使用下面语句创建输出列表：

```
(list 'if test
  (cons 'begin branch))
```

其中test与宏的第一个子`form`绑定，即：

```
(< (pressure tube) 60)
```

同时`branch`与余下的宏的子`form`绑定，即：

```
((open-valve tube)
 (attach floor-pump tube)
 (depress floor-pump 5)
 (detach floor-pump tube)
 (close-valve tube))
```

输出列表可能会变得相当复杂。我们很容易能够发现比when更加庞大的宏可以对输出列表完成精心的加工工程。这种情况下，更方便的方法是把宏的输出指定为模板，对宏的每种用法把相关参数插入到模板的适当位置。Scheme提供了backquote语法来指定这种模板。因此表达式：

```
(list 'IF test
  (cons 'BEGIN branch))
```

写成这样会更加方便：

```
`(IF ,test
  (BEGIN ,@branch))
```

我们能够将 `when` 的宏表达式重构为：

```
(define-macro when
  (lambda (test . branch)
    `(IF ,test
        (BEGIN ,@branch))))
```

注意模板的格式，并不像早先列表的结构，而是对输出列表的形态给出了直接的视觉指示。反引号（`）为列表引进了一个模板。除了以逗号（,）或（,@）作为前缀的元素外，模板的元素会在结果列表中逐字出现。（为了举例，我们把模板的每一个会在结果中原封不动出现元素写成了大写）。

`,` 和 `,@` 可以将宏参数插入到模板中。`,` 插入的是逗号后面紧接着它的下一个表达式求值后的结果。`,@` (comma-splice)插入的是它的下一个表达式先splice再求值的结果。即：它消除了最外面的括号。（这说明被comma-splice引用的表达式必须是一个列表。）

在我们的例子中，给定 `test` 和 `branch` 的绑定值，很容易看到模板将扩展到所需的地步。

```
(IF (< (pressure tube) 60)
    (BEGIN
      (open-valve tube)
      (attach floor-pump tube)
      (depress floor-pump 5)
      (detach floor-pump tube)
      (close-valve tube)))
```

8.2 避免在宏内部产生变量捕获

一个二变量的 `disjunction form` ，`my-or` ，可以定义为：

```
(define-macro my-or
  (lambda (x y)
`(if ,x ,x ,y)))
```

`my-or` 带有两个参数并返回两个之中第一个为真（非#f）的值。特别的，只有当第一个参数为假时才会对第二个参数求值。

```
(my-or 1 2)
=>  1
(my-or #f 2)
=>  2
```

上述的 `my-or` 宏时会有一个问题。如果第一个参数为真，会重新求值第一个参数：第一次是在if语句中，第二次在then分支。如果第一个参数包含副作用，这会造成意外的结果，例如：

```
(my-or
  (begin
    (display "doing first argument")
     (newline)
     #t)
  2)
```

会显示 `doing first argument` 两次。

这个情况可以通过在局部变量中储存if测试结果来避免：

```
(define-macro my-or
  (lambda (x y)
    `(let ((temp ,x))
      (if temp temp ,y))))
```

这样基本上OK了，除非当第二个参数在宏定义中使用时包含相同的temp。例如：

```
(define temp 3)

(my-or #f temp)
=>  #f
```

当然结果应该是3！错误产生的原因是由于宏使用了局部变量 `temp` 储存第一个参数（ `#f` ）的值，而第二个参数中的变量 `temp` 被宏引入的 `temp` 所捕获。

```
(define temp 3)

(let ((temp #f))
  (if temp temp 3))
```

为避免这类错误，我们在选择宏定义中的局部变量时需要小心行事。我们应该为这些变量选择古怪的名字并热切希望没有人会跟它们扯上关系。例如：

```
(define-macro my-or
  (lambda (x y)
    `(let ((+temp ,x))
       (if +temp +temp ,y))))
```

如果默认+temp在宏之外的代码中不被使用，则它就是正确的。但这种幻想是迟早要破灭的。

一个更加可靠详细的方法就是生成保证不会被其他方式占用的符号。当调用 `gensym` 程序时，它会产生出独一无二的标志。这是一个使用 `gensym` 的 `my-or` 的安全定义：

```
(define-macro my-or
  (lambda (x y)
    (let ((temp (gensym)))
      `(let ((,temp ,x))
         (if ,temp ,temp ,y)))))
```

为了简明，在本文中定义的宏，不使用 `gensym` 方法。相反，我们将假设变量捕获这个问题已经被考虑到了，而使用更加简明的 `+` 作为前缀。我们把这些将加号开头的标识符转换为gensym的工作留给敏锐的读者。

8.3 fluid-let

这有一个更加复杂的宏的定义， `fluid-let` （见5.2节）。 `fluid-let` 对一组已经存在的词法变量指定了临时绑定。假定一个fluid-let表达式如下：

```
(fluid-let ((x 9) (y (+ y 1)))
  (+ x y))
```

我们想扩展为：

```
(let ((OLD-X x) (OLD-Y y))
  (set! x 9)
  (set! y (+ y 1))
  (let ((RESULT (begin (+ x y))))
    (set! x OLD-X)
    (set! y OLD-Y)
    RESULT))
```

在例子中我们希望标识符 `OLD-X` ， `OLD-Y` 和 `RESULT` 不会捕获 `fluid-let` 里的变量。

下述例子教你如何构造一个可以实施你的想法的 `fluid-let` 宏：

```
(define-macro fluid-let
  (lambda (xexe . body)
    (let ((xx (map car xexe))
          (ee (map cadr xexe))
          (old-xx (map (lambda (ig) (gensym)) xexe))
          (result (gensym)))
      `(let ,(map (lambda (old-x x) `(,old-x ,x))
                  old-xx xx)
         ,@(map (lambda (x e)
                  `(set! ,x ,e))
                xx ee)
         (let ((,result (begin ,@body)))
           ,@(map (lambda (x old-x)
                    `(set! ,x ,old-x))
                  xx old-xx)
           ,result)))))
```

宏的参数是 `xexe` ，是由 `fluid-let` 引进的变量/表达式列表；而 `body` ，则是在 `fluid-let` 主体中的表达式列表。在我们的例子中，这两者分别是 `((x 9) (y (+ y 1)` 和 `((+ xy))` 。

宏的主体引进了一堆局部变量： `xx` 是从变量/表达式中提取的变量列表。 `ee` 是对应的表达式列表。 `old-xx` 是新的标识符的列表，对应于 `xx` 中的每个变量。这些曾用来储存 `xx` 的传入值，这样我们可以将 `xx` 恢复到 `fluid-let` 主体求值前的状态。Result是另一个新标志符，用来储存 `fluid-let` 主体的值。在我们的例子中， `xx` 是 `(x y)` ， `ee` 是 `(9(+ y 1))` 。根据你的系统实现 `gensym` 的方式， `old-xx` 会成为列表 `(GEN-63 GEN-64)` ， `result` 会成为 `GEN-65` 。

在我们的例子中，由宏创建的输出列表像这样：

```
(let ((GEN-63 x) (GEN-64 y))
  (set! x 9)
  (set! y (+ y 1))
  (let ((GEN-65 (begin (+ x y))))
    (set! x GEN-63)
    (set! y GEN-64)
    GEN-65))
```

这确实可以满足我们的需求。

第九章 结构

自然分组的数据被称为结构。我们可以使用Scheme提供的复合数据结构如向量和列表来表示一种“结构”。例如：我们正在处理与树木相关的一组数据。数据（或者叫字段 `field` ）中的单个元素包括：高度，周长，年龄，树叶形状和树叶颜色共5个字段。这样的数据可以表示为5元向量。这些字段可以利用 `vector-ref` 访问，或使用 `vector-set!` 修改。尽管如此，我们仍然不希望记忆向量索引编号与字段的对于关系，这将是一个费力不讨好而且容易出错的事情，尤其是随着时间的流逝，一些字段被加进来，而另一些字段会被删掉。

因此我们使用Scheme的宏 `defstruct` 去定义一个 `结构` ，基本上你可以把它当作一种向量，不过它提供了很多方法诸如创建 `结构` 实例、访问或修改它的字段等等。因此，我们的树结构应这样定义：

```
(defstruct tree height girth age leaf-shape leaf-color)
```

这样它自动生成了一个名为 `make-tree` 的构造过程，以及每个字段的访问方法，命名为 `tree.height` ， `tree.girth` 等等。构造方法的使用方法如下：

```
(define coconut
  (make-tree 'height 30
             'leaf-shape 'frond
             'age 5))
```

这个构造函数的参数以成对的形式出现，字段名后面紧跟着其初始值。这些字段能以任意顺序出现，或者不出现——如果字段的值没有定义的话。

访问过程的调用如下所示：

```
(tree.height coconut) =>  30
(tree.leaf-shape coconut) =>  frond
(tree.girth coconut) =>  <undefined>
```

`tree.girth` 存取程序返回一个未定义的值，因为我们没有为 `coconut` 这个 `tree` 结构指定 `girth` 的值。

修改过程的调用如下所示：

```
(set!tree.height coconut 40)
(set!tree.girth coconut 10)
```

如果我们现在重新调用访问过程去访问这些字段，我们会得到新的值：

```
(tree.height coconut) =>  40
(tree.girth coconut) =>  10
```

9.1 默认初始化

我们可以在定义结构时进行一些初始化的设置，而不是在每个实例中都进行初始化。因此，我们假定 `leaf-shape` 和 `leaf-color` 在默认情况下分别为 `frond` 和 `green` 。我们可以在调用make-tree时通过显式的初始化来覆盖掉这些默认值，或者在创建一个结构实例后使用上面提到的字段修改过程：

```
(defstruct tree height girth age
                (leaf-shape 'frond)
                (leaf-color 'green))

(define palm (make-tree 'height 60))

(tree.height palm)
=>  60

(tree.leaf-shape palm)
=>  frond

(define plantain
  (make-tree 'height 7
             'leaf-shape 'sheet))

(tree.height plantain)
=>  7

(tree.leaf-shape plantain)
=>  sheet

(tree.leaf-color plantain)
=>  green
```

9.2 defstruct定义

宏 `defstruct` 的定义如下：

```
(define-macro defstruct
  (lambda (s . ff)
    (let ((s-s (symbol->string s)) (n (length ff)))
      (let* ((n+1 (+ n 1))
             (vv (make-vector n+1)))
        (let loop ((i 1) (ff ff))
          (if (<= i n)
```

```
              (let ((f (car ff)))
                (vector-set! vv i
                  (if (pair? f) (cadr f) '(if #f #f)))
                (loop (+ i 1) (cdr ff))))))
          (let ((ff (map (lambda (f) (if (pair? f) (car f) f))
                         ff)))
            `(begin
               (define ,(string->symbol
                         (string-append "make-" s-s))
                 (lambda fvfv
                   (let ((st (make-vector ,n+1)) (ff ',ff))
                     (vector-set! st 0 ',s)
                     ,@(let loop ((i 1) (r '()))
                         (if (>= i n+1) r
                             (loop (+ i 1)
                                   (cons `(vector-set! st ,i
                                                       ,(vector-ref vv i))
                                         r))))
                     (let loop ((fvfv fvfv))
                       (if (not (null? fvfv))
                           (begin
                             (vector-set! st
                                 (+ (list-position (car fvfv) ff)
                                    1)
                               (cadr fvfv))
                             (loop (cddr fvfv)))))
                     st)))
               ,@(let loop ((i 1) (procs '()))
                   (if (>= i n+1) procs
                       (loop (+ i 1)
                             (let ((f (symbol->string
                                       (list-ref ff (- i 1)))))
                               (cons
                                `(define ,(string->symbol
                                            (string-append
                                             s-s "." f))
                                   (lambda (x) (vector-ref x ,i)))
                                (cons
                                 `(define ,(string->symbol
                                             (string-append
                                              "set!" s-s "." f))
                                    (lambda (x v)
                                      (vector-set! x ,i v)))
                                 procs))))))
               (define ,(string->symbol (string-append s-s "?"))
                 (lambda (x)
                   (and (vector? x)
                        (eqv? (vector-ref x 0) ',s))))))))))
```

第十章 关联表和表格

关联表是Scheme一种特殊形式的列表。列表的每一个元素都是一个点对，其中的car（左边的元素）被称为一个“键”，cdr（右边的元素）被称为和该键关联的值。例如：

```
((a . 1) (b . 2) (c . 3))
```

调用程序 `(assv k al)` 能在关联表 `al` 中找到和键 `k` 关联的CONS单元。在查找时关联表中的键与 `k` 使用 `eqv?` 过程来比较。然而有时我们可能希望自定义一个键的比较函数。例如，如果键是不区分大小写的字符串，那默认的 `eqv?` 就没什么用了。

我们现在定义一个结构 `table` (表格)，这是一个改进后的关联表，它可以允许用户在它的键上自定义比较函数。它的字段是 `equ` 和 `alist` 。

```
(defstruct table (equ eqv?) (alist '()))
```

（默认的比较函数是 `eqv?` ——对于一个普通的关联表——关联表的初始化为空。）

我们将使用程序 `table-get` 得到与一个给定键关联的值（相对于cons单元）。 `table-get` 接受一个 `table` (表格)和一个键作为参数，还有一个可选的默认值，这样若在表格中未找到该键则返回该默认值：

```
(define table-get
  (lambda (tbl k . d)
    (let ((c (lassoc k (table.alist tbl) (table.equ tbl))))
      (cond (c (cdr c))
            ((pair? d) (car d))))))
```

在 `table-get` 中使用的程序 `lassoc` ，定义如下：

```
(define lassoc
  (lambda (k al equ?)
    (let loop ((al al))
      (if (null? al) #f
          (let ((c (car al)))
            (if (equ? (car c) k) c
                (loop (cdr al))))))))
```

程序 `table-put` 用来更新给定表格中的一个键的值：

```
(define table-put!
  (lambda (tbl k v)
    (let ((al (table.alist tbl)))
      (let ((c (lassoc k al (table.equ tbl))))
        (if c (set-cdr! c v)
            (set!table.alist tbl (cons (cons k v) al)))))))
```

程序 `table-for-each` 为每个表格中键/值对调用给定的程序

```
(define table-for-each
  (lambda (tbl p)
    (for-each
     (lambda (c)
       (p (car c) (cdr c)))
     (table.alist tbl))))
```

第十一章 系统接口

一个有用的Scheme程序经常需要与底层操作系统进行交互。

11.1 检查和删除文件

`file-exists?` 会检查它的参数字符串是否是一个文件。`delete-file` 接受一个文件名字符串作为参数并删除相应的文件。这些程序并不是Scheme标准的一部分，但是在大多数Scheme实现中都能找到它们。用这些过程操作目录（而不是文件）并不是很可靠。（用它们操作目录的结果与具体的Scheme实现有关。）

`file-or-directory-modify-seconds` 过程接受一个文件名或目录名为参数，并返回这个目录或文件的最后修改时间。时间是从格林威治标准时间1970年1月1日0点开始记时的。例如：

```
(file-or-directory-modify-seconds "hello.scm")
=>  893189629
```

假定 `hello.scm` 文件最后一次修改的时间是1998年4月21日的某个时间。

11.2 调用操作系统命令

`system` 程序把它的参数字符串当作操作系统命令来执行 [1]。如果命令成功执行并返回0，则它会返回真，如果命令执行失败并返回某非0值，则它会返回假。命令产生的任何输出都会进入标准的输出。

```
(system "ls")
;lists current directory

(define fname "spot")

(system (string-append "test -f " fname))
;tests if file `spot' exists

(system (string-append "rm -f " fname))
;removes `spot'
```

最后两个命令等价于：

```
(file-exists? fname)

(delete-file fname)
```

11.3 环境变量

过程 `getenv` 返回操作系统环境变量的设定值，如：

```
(getenv "HOME")
=>  "/home/dorai"

(getenv "SHELL")
=>  "/bin/bash"
```

---

[1] MzScheme在 `process` 库中提供了 `system` 过程。使用 `(require (lib "process.ss"))` 来加载这个库。

第十二章 对象和类

类是描述了一组有共同行为的对象。由类描述的对象称为类的一个实例。类指定了其实例拥有的 `属性` （原文为slot卡槽）的名称，而这些 `属性` 的值由实例自身来进行填充。类同样也指定了可以应用于其实例的 `方法` (method)。属性值可以是任何形式，但方法的值必须是过程。

类具有继承性。因此，一个类可以是另一个类的子类，我们称另一个类为它的父类。一个子类不仅有它自己“直接的”属性和方法，也会继承它的父类的所有属性和方法。如果一个类里有与其父类相同名称的属性和方法，那么仅保留子类的属性和方法。

12.1 一个简单的对象系统

现在我们用Scheme来实现一个基本的对象系统。对于每个类，我们只允许有一个父类（单继承性）。如果我们不想指定一个父类，我们可以用 `#t` 作为一个“元”父类，既没有属性，也没有方法。而 `#t` 的父类则认为是它自己。

作为一次尝试，用结构 `standard-class` 来定义类应该是很好的一种方式，用结构的字段来保存属性名字，父类以及方法。前两个字段我们分别叫做 `slots` 和 `superclass` 。我们将使用两个字段来描述方法，用 `method-names` 字段来描述类的方法的名称列表，用 `method-vector` 字段来保存一个矢量，里面放着类的方法。这是 `standard-class` 的定义：

```
(defstruct standard-class
  slots superclass method-names method-vector)
```

我们可以用 `make-standard-class` ，即 `standard-class` 的制造程序(见第九章)来创建一个新的类：

```
(define trivial-bike-class
  (make-standard-class
   'superclass #t
   'slots '(frame parts size)
   'method-names '()
   'method-vector #()))
```

这是一个非常简单的类，更加复杂的类会有有意义的父类和方法，这需要在创建类时进行大量的初始化设置，我们希望把这些工作隐藏在创建类的过程中。因此我们定义一个 `create-class` 宏来对 `make-standard-class` 进行适当的调用。

```
(define-macro create-class
  (lambda (superclass slots . methods)
    `(create-class-proc
      ,superclass
      (list ,@(map (lambda (slot) `',slot) slots))
      (list ,@(map (lambda (method) `',(car method)) methods))
      (vector ,@(map (lambda (method) `,(cadr method)) methods)))))
```

我们稍后再介绍 `create-class-proc` 程序的定义。

`make-instance` 程序创建类的一个实例，由类中包含的信息产生一个新的向量。实例向量的格式非常简单：它的第一个元素指向这个类（引用），余下的元素都是属性值。 `make-instance` 的第一个参数是一个类，后面的参数是成对的序列，而每一个“对”是属性名称和该实例中属性的值。

```
(define make-instance
  (lambda (class . slot-value-twosomes)

    ;Find `n', the number of slots in `class'.
    ;Create an instance vector of length `n + 1',
    ;because we need one extra element in the instance
    ;to contain the class.

    (let* ((slotlist (standard-class.slots class))
           (n (length slotlist))
           (instance (make-vector (+ n 1))))
      (vector-set! instance 0 class)

      ;Fill each of the slots in the instance
      ;with the value as specified in the call to
      ;`make-instance'.

      (let loop ((slot-value-twosomes slot-value-twosomes))
        (if (null? slot-value-twosomes) instance
            (let ((k (list-position (car slot-value-twosomes)
                                    slotlist)))
              (vector-set! instance (+ k 1)
                (cadr slot-value-twosomes))
              (loop (cddr slot-value-twosomes))))))))
```

这是一个类的实例化的例子：

```
(define my-bike
  (make-instance trivial-bike-class
                 'frame 'cromoly
                 'size '18.5
                 'parts 'alivio))
```

这将 `my-bike` 变量绑定到如下所示的实例上。

```
#(<trivial-bike-class> cromoly 18.5 alivio)
```

`<trivial-bike-class>` 是一个Scheme数据(另一个向量)代表之前定义的 `trivia-bike-class` 的值。

`class-of` 程序返回该实例对应的类：

```
(define class-of
  (lambda (instance)
    (vector-ref instance 0)))
```

这里假定 `class-of` 的参数是一个类的实例，即一个向量，其第一个元素指向 `standard-class` 的一些实例。我们可能想使 `class-of` 对我们给定的任何类型Scheme对象返回一个合适的值。

```
(define class-of
  (lambda (x)
    (if (vector? x)
        (let ((n (vector-length x)))
          (if (>= n 1)
              (let ((c (vector-ref x 0)))
                (if (standard-class? c) c #t))
              #t))
        #t)))
```

不是用 `standard-class` 创建的Scheme对象的类被认为是 `#t` ，即“元类”。

`slot-value` 过程和 `set!slot-value` 过程用来访问和改变一个类实例的值：

```
(define slot-value
  (lambda (instance slot)
    (let* ((class (class-of instance))
           (slot-index
            (list-position slot (standard-class.slots class))))
      (vector-ref instance (+ slot-index 1)))))

(define set!slot-value
  (lambda (instance slot new-val)
    (let* ((class (class-of instance))
           (slot-index
            (list-position slot (standard-class.slots class))))
      (vector-set! instance (+ slot-index 1) new-val))))
```

我们现在来解决`create-class-proc`的定义问题。这个过程接受一个父类，一个属性的列表，一个方法名称的列表和一个包含方法体的向量，并适当调用`make-standard-class`程序。唯一困难的部分是给定的属性字段的值。由于一个类必须包括它的父类的属性，因此不能只有`create-class`提供的属性参数。我们必须把所给的属性追加到父类的属性中，并保证没有重复的属性。

```
(define create-class-proc
  (lambda (superclass slots method-names method-vector)
    (make-standard-class
     'superclass superclass
     'slots
     (let ((superclass-slots
            (if (not (eqv? superclass #t))
                (standard-class.slots superclass)
                '())))
       (if (null? superclass-slots) slots
           (delete-duplicates
            (append slots superclass-slots))))
     'method-names method-names
     'method-vector method-vector)))
```

过程`delete-duplicates`接受一个列表`s`为参数，返回一个新列表，该列表只包含`s`中每个元素的最后一次出现。

```
(define delete-duplicates
  (lambda (s)
    (if (null? s) s
        (let ((a (car s)) (d (cdr s)))
          (if (memv a d) (delete-duplicates d)
              (cons a (delete-duplicates d)))))))
```

现在谈谈方法的应用。我们通过使用`send`程序调用一个类实例的方法。`send`的参数是方法的名字，紧接着是类实例，以及除了类实例本身之外的该方法的其他参数。由于方法储存在实例的类中而不是在实例本身中，因此`send`会在该实例对于的类中寻找该方法。如果没有找到，则到父类中寻找，如此直到找完整个继承链：

```
(define send
  (lambda (method instance . args)
    (let ((proc
           (let loop ((class (class-of instance)))
             (if (eqv? class #t) (error 'send)
                 (let ((k (list-position
                           method
                           (standard-class.method-names class))))
                   (if k
                       (vector-ref (standard-class.method-vector cl
                       (loop (standard-class.superclass class))))))
      (apply proc instance args))))
```

我们现在可以定义一些更有趣的类了：

```
(define bike-class
  (create-class
   #t
   (frame size parts chain tires)
   (check-fit (lambda (me inseam)
                (let ((bike-size (slot-value me 'size))
                      (ideal-size (* inseam 3/5)))
                  (let ((diff (- bike-size ideal-size)))
                    (cond ((<= -1 diff 1) 'perfect-fit)
                          ((<= -2 diff 2) 'fits-well)
                          ((< diff -2) 'too-small)
                          ((> diff 2) 'too-big))))))))
```

这里，`bike-class` 包括一个名为 `check-fit` 的方法，它接受一个自行车的实例和一个裤腿的尺寸作为参数，并报告该车对这种裤腿尺寸的人的适应性。

我们再来定义 `my-bike` ：

```
(define my-bike
  (make-instance bike-class
                 'frame 'titanium ; I wish
                 'size 21
                 'parts 'ultegra
                 'chain 'sachs
                 'tires 'continental))
```

检查这个车与裤腿尺寸为32的某个人是否搭配：

```
(send 'check-fit my-bike 32)
```

我们再定义子类 `bike-class` 。

```
(define mtn-bike-class
  (create-class
    bike-class
    (suspension)
    (check-fit (lambda (me inseam)
                 (let ((bike-size (slot-value me 'size))
                       (ideal-size (- (* inseam 3/5) 2)))
                   (let ((diff (- bike-size ideal-size)))
                     (cond ((<= -2 diff 2) 'perfect-fit)
                           ((<= -4 diff 4) 'fits-well)
                           ((< diff -4) 'too-small)
                           ((> diff 4) 'too-big))))))))
```

`Mtn-bike-class` 添加了一个名为 `suspension` 的属性。并定义了一个稍微不同的名为 `check-fit` 的方法。

12.2 类也是实例

到这里为止，精明的读者可能已经发现了：类本身可以是某些其他类（如“元类”）的实例。注意所有类都有一些相同的特点：每个都有属性、父类、方法名称的列表和包含方法体的向量。 `make-instance` 看起来像是他们所共享的方法。这意味着我们可以通过另一个类（当然也是某个类的实例啦）来指定这些共同的特点。

具体的说就是我们可以重写我们的类实现并实现其自身（好别扭）。使用面向对象的方法，这样我们可以确保不会遇到鸡生蛋，蛋生鸡的问题。这样我们会跳出 `class` 结构和它相关的过程并余下的方法来把类定义为对象。

我们现在把 `standard-class` 作为其他类的父类。特别的， `standard-class` 必须是它自己的一个实例。那么 `standard-class` 应该是什么样子的呢？

我们知道 `standard-class` 是一个实例，而且我们用一个向量来表示这个实例。所以最终是一个向量，其第一个元素是它的父类，也就是它自己，而余下的元素是属性值。我们已经确定有四个所有类都必须有的属性，因此 `standard-class` 是一个5个元素的向量。

```
(define standard-class
  (vector 'value-of-standard-class-goes-here
          (list 'slots
                'superclass
                'method-names
                'method-vector)
          #t
          '(make-instance)
          (vector make-instance)))
```

注意到 `standard-class` 这个向量并没有被完全填充：符号 `value-of-standard-class-goes-here` 此时仅仅做占位用。现在我们已经定义了一个 `standard-class` 的值，现在我们可以用它来确定它自己的类，即它本身。

```
(vector-set! standard-class 0 standard-class)
```

注意我们不能用 `class` 结构提供的过程了。我们必须把下面的形式：

```
(standard-class? x)
(standard-class.slots c)
(standard-class.superclass c)
(standard-class.method-names c)
(standard-class.method-vector c)
(make-standard-class ...)
```

换成：

```
(and (vector? x) (eqv? (vector-ref x 0) standard-class))
(vector-ref c 1)
(vector-ref c 2)
(vector-ref c 3)
(vector-ref c 4)
(send 'make-instance standard-class ...)
```

12.3 多重继承

我们可以容易的修改这个对象系统使类可以有一个以上的父类。我们重新定义 `standard-class` 来添加一个属性叫 `class-precedence-list` 取代 `superclass` ，一个类的 `class-precedence-list` 是它所有父类的列表，而不只有通过 `create-class` 创建时指定的“直接”的父类。从这个名字可以看出其超类是以一种特定的顺序来存放的，前面的超类有比后面超类更高的优先级。

```
(define standard-class
  (vector 'value-of-standard-class-goes-here
          (list 'slots 'class-precedence-list 'method-names 'methoc
          '()
          '(make-instance)
          (vector make-instance)))
```

不仅属性列表改变来存放新的属性，而且 `superclass` 属性也从 `#t` 变为 `()` ，这是因为 `standard-class` 的 `class-precedence-list` 必须是一个列表。我们可以令它的值为 `(#t)` ，但是我们不会提到元类，由于它在每个类

的 `class-precedence-list` 中。

宏 `create-class` 也需要修改来接受一个超类的列表而不是一个单独的超类。

```
(define-macro create-class
  (lambda (direct-superclasses slots . methods)
    `(create-class-proc
      (list ,@(map (lambda (su) `,su) direct-superclasses))
      (list ,@(map (lambda (slot) `',slot) slots))
      (list ,@(map (lambda (method) `',(car method)) methods))
      (vector ,@(map (lambda (method) `,(cadr method)) methods))
      )))
```

`create-class-proc` 必须根据提供的超类给出类的优先级列表，并根据优先级给出属性列表：

```
(define create-class-proc
  (lambda (direct-superclasses slots method-names method-vector)
    (let ((class-precedence-list
           (delete-duplicates
            (append-map
             (lambda (c) (vector-ref c 2))
             direct-superclasses))))
      (send 'make-instance standard-class
            'class-precedence-list class-precedence-list
            'slots
            (delete-duplicates
             (append slots (append-map
                            (lambda (c) (vector-ref c 1))
                            class-precedence-list)))
            'method-names method-names
            'method-vector method-vector))))
```

过程 `append-map` 是一个 `append` 和 `map` 的组合：

```
(define append-map
  (lambda (f s)
    (let loop ((s s))
      (if (null? s) '()
          (append (f (car s))
                  (loop (cdr s)))))))
```

过程 `send` 在寻找一个方法时必须从左到右搜索类的优先级列表：

```
(define send
  (lambda (method-name instance . args)
    (let ((proc
           (let ((class (class-of instance)))
             (if (eqv? class #t) (error 'send)
                 (let loop ((class class)
                            (superclasses (vector-ref class 2)))
                   (let ((k (list-position
                             method-name
                             (vector-ref class 3))))
                     (cond (k (vector-ref
                               (vector-ref class 4) k))
                           ((null? superclasses) (error 'send))
                           (else (loop (car superclasses)
                                       (cdr superclasses))))
                     )))))))
      (apply proc instance args))))
```

理论上我们可以把方法也定义为属性（值为一个过程），但是有很多理由不这样做，类的实例共享方法但是通常有不同的属性值。也就是说，方法可以包括在类定义中，而且不需要每次实例化时都进行设置——就像属性那样。

第十三章 跳转

Scheme的一个显著标志是它支持跳转或者`nonlocal control`。特别是Scheme允许程序控制跳转到程序的任意位置，相比之下条件语句和函数调用的限制要更多一些。Scheme的`nonlocal control`操作符是一个名为`call-with-current-continuation`的过程。下面我们会看到如何用这个操作符创建一些惊人的控制效果。

13.1 call-with-current-continuation

`call-with-current-continuation`用`current-continuation`来调用它的参数（一个只有一个参数的过程）【在调用时传入参数`current-continuation`，译者注】。这就是这个操作符名字的解释了。但是由于这个名字太长，故通常缩写为`call/cc`[1]。

一个程序执行到任意一点的当前续延【即`current-continuation`，译者注】是该程序的后半部分【即将要被执行的部分，译者注】。因此在程序：

```
(+ 1 (call/cc
       (lambda (k)
         (+ 2 (k 3)))))
```

中，“后边部分”——从`call/cc`程序的角度来看，是如下的带有一个“洞”的程序（“洞”用`[]`表示）：

```
(+ 1 [])
```

也就是说，该程序的“续延”是一个把`1`和填到这个“洞”里的东西加起来的程序。

这就是`call/cc`参数被调用的情况。记住`call/cc`的参数是过程：

```
(lambda (k)
  (+ 2 (k 3)))
```

这个过程的把“续延”（现在绑定在`k`上）`apply`到参数`3`上。这就是这个续延与众不同之处。“续延”调用突然放弃了它自己的计算并把当前的计算换成了`k`中保存的程序。也就是说，这个程序中加`2`的操作被放弃了，然后`k`的参数`3`直接被发送到了那个带“洞”的程序：

```
(+ 1 [])
```

然后程序就简单的变成：

```
(+ 1 3)
```

然后返回 `4` 。即：

```
(+ 1 (call/cc
       (lambda (k)
         (+ 2 (k 3)))))
```

上面的例子叫做"'退出'续延"，用来退出某个计算过程（这里是 `(+ 2 [])` 的计算）。这是一个很有用的功能，但是Scheme的续延可以用来返回到前面放弃计算的地方，然后多次调用它们。程序的"后半部分"意味着一个续延不论我们调用的次数和时间都存在，这也让 `call/cc` 更加强大和令人迷惑。看一个简单的例子，在解释器里输入以下代码：

```
(define r #f)

(+ 1 (call/cc
       (lambda (k)
         (set! r k)
         (+ 2 (k 3)))))
=>  4
```

后面的表达式和刚才一样返回了 `4` ，不同之处在于这次我们把续延 `k` 保存到了全局变量 `r` 里。

现在我们在 `r` 中永久保存了这个续延。如果我们以一个数字为参数调用它，就会返回数字加1后的结果：

```
(r 5)
=>  6
```

注意 `r` 会放弃它自己的续延，打个比方我们把对 `r` 的调用放在一个上下文中：

```
(+ 3 (r 5))
=>  6
```

因此 `call/cc` 提供的续延是一种"放弃"的续延。

13.2 "退出"续延

“退出”续延是 `call/cc` 最简单的用法，而且在退出函数或循环时非常有用。考虑一个过程 `list-product` 接收一个数字列表并把所有的数乘起来。一个直观的递归定义可以这样写：

```
(define list-product
  (lambda (s)
    (let recur ((s s))
      (if (null? s) 1
          (* (car s) (recur (cdr s)))))))
```

这个方法有一个问题。如果列表中有一个数是0，而且0后面还有很多元素，那么结果是可以预知的。如果这样上面的代码会在得出结果前产生很多无意义的递归调用。这就是“退出”续延大显身手的时候。用 `call/cc` ，我们可以这样重写这个过程：

```
(define list-product
  (lambda (s)
    (call/cc
      (lambda (exit)
        (let recur ((s s))
          (if (null? s) 1
              (if (= (car s) 0) (exit 0)
                  (* (car s) (recur (cdr s))))))))))
```

如果遇到一个为0的元素，续延 `exit` 就会以参数0被调用，这样就防止了更多的调用 `recur` 。

13.3 树匹配

一个更加复杂的例子是把续延用于解决两个树是否有相同边缘（就是相同的元素（叶节点）有相同的顺序）的问题上。如：

```
(same-fringe? '(1 (2 3)) '((1 2) 3))
=>  #t

(same-fringe? '(1 2 3) '(1 (3 2)))
=>  #f
```

纯粹的函数式解决方案是把两个树都抹平然后看结果是否一样。

```
(define same-fringe?
  (lambda (tree1 tree2)
    (let loop ((ftree1 (flatten tree1))
               (ftree2 (flatten tree2)))
      (cond ((and (null? ftree1) (null? ftree2)) #t)
            ((or (null? ftree1) (null? ftree2)) #f)
            ((eqv? (car ftree1) (car ftree2))
             (loop (cdr ftree1) (cdr ftree2)))
            (else #f)))))

(define flatten
  (lambda (tree)
    (cond ((null? tree) '())
          ((pair? (car tree))
           (append (flatten (car tree))
                   (flatten (cdr tree))))
          (else
           (cons (car tree)
                 (flatten (cdr tree)))))))
```

然而，这样会遍历整个树来进行抹平操作，而且还要再做一遍这样的操作【遍历被抹平后生成的列表，译者注】才能找到不匹配的元素。退一步讲，即使最好的抹平算法也需要 `cons` （直接修改输入的树是不可以的）

我们可以用 `call/cc` 来解决这个问题，不需要遍历，也不需要用 `cons` 来拼接。每个树会被 `map` 到一个生成器——一个带有内部状态的过程，按照叶节点在树中出现的顺序从左到右连续的输出叶节点。

```
(define tree->generator
  (lambda (tree)
    (let ((caller '*))
      (letrec
          ((generate-leaves
            (lambda ()
              (let loop ((tree tree))
                (cond ((null? tree) 'skip)
                      ((pair? tree)
                       (loop (car tree))
                       (loop (cdr tree)))
                      (else
                       (call/cc
                        (lambda (rest-of-tree)
                          (set! generate-leaves
                            (lambda ()
                              (rest-of-tree 'resume)))
                          (caller tree))))))
              (caller '()))))
        (lambda ()
          (call/cc
           (lambda (k)
             (set! caller k)
             (generate-leaves))))))))
```

当一个 `tree->generator` 创建的生成器被调用时，这个生成器会把调用的续延存在 `caller` 中，这样它就知道当找到叶节点时把它发送给谁。然后它调用一个内部定义的函数 `generate-leaves` ，该函数会从左到右循环遍历这个树。当循环到一个叶节点时，该函数就使用 `caller` 来返回该叶节点作为生成器的结果，但是它会记住后续的循环（被 `call/cc` 捕获为一个续延）并保存到 `generate-leaves` 变量，下次生成器被调用时，循环从刚才终端的地方恢复，这样它可以寻找下一个叶节点。

注意 `generate-leaves` 做的最后一件事情，在循环结束后，它返回一个空列表给 `caller` 。由于空列表不是一个合法的叶节点，我们可以用它来告诉生成器没有叶节点需要生成了。

过程 `same-fringe?` 把树作为参数来创建生成器，然后交替调用这两个生成器。只要一找到两个不同的叶节点就会返回失败。

```
(define same-fringe?
  (lambda (tree1 tree2)
    (let ((gen1 (tree->generator tree1))
          (gen2 (tree->generator tree2)))
      (let loop ()
        (let ((leaf1 (gen1))
              (leaf2 (gen2)))
          (if (eqv? leaf1 leaf2)
              (if (null? leaf1) #t (loop))
              #f))))))
```

很容易看到每个树只被遍历了最多一次，在遇到不匹配的情况时，只会遍历最左边的那个不匹配节点【？】。而且没有用到 `cons` 。

13.4 协程

上面用到的生成器是一些有趣而普遍的过程概念。每次生成器被调用时，它都恢复计算，而且当它返回前会把它的续延保存在一个内部变量中这样这个生成器可以再次恢复。我们可以对生成器进行推广，这样他们可以相互恢复其他的生成器，并且互相传递结果。这样的过程叫协程。

我们将会看到一个协程作为一元过程，其主体可以包含 `resume` 调用， `resume` 是一个两参数的过程，可以被一个协程用来继续执行另一个协程（带着一个转换值）。宏 `coroutine` 定义一个这样的协程过程，一个变量名作为协程的初始参数，内容作为协程。

```
(define-macro coroutine
  (lambda (x . body)
    `(letrec ((+local-control-state (lambda (,x) ,@body))
              (resume
                (lambda (c v)
                 (call/cc
                  (lambda (k)
                    (set! +local-control-state k)
                    (c v))))))
       (lambda (v)
         (+local-control-state v)))))
```

调用这个宏可以创建一个协程（我们叫为 `A` ），这个协程可以有一个参数。 `A` 有一个内部变量叫做 `+local-control-state` 来保存任意时刻这个协程接下来的计算。当调用 `resume` 时——也就是调用另一个协程 `B` 时——当前协程会更新它的 `+local-control-state` 变量为之后的计算，然后停止，然后跳到恢复了的协程 `B` ，当协程 `A` 之后恢复时，它的计算会从它 `+local-control-state` 变量里存放的续延开始。

13.4.1 用协程进行树匹配

用协程会进一步简化树匹配的操作。匹配过程被编写为一个协程，该协程依赖另外两个协程提供各自的叶节点。

```
(define make-matcher-coroutine
  (lambda (tree-cor-1 tree-cor-2)
    (coroutine dont-need-an-init-arg
      (let loop ()
        (let ((leaf1 (resume tree-cor-1 'get-a-leaf))
              (leaf2 (resume tree-cor-2 'get-a-leaf)))
          (if (eqv? leaf1 leaf2)
              (if (null? leaf1) #t (loop))
              #f))))))
```

叶生成器协程会记住把它的节点返回给谁：

```
(define make-leaf-gen-coroutine
  (lambda (tree matcher-cor)
    (coroutine dont-need-an-init-arg
      (let loop ((tree tree))
        (cond ((null? tree) 'skip)
              ((pair? tree)
               (loop (car tree))
               (loop (cdr tree)))
              (else
               (resume matcher-cor tree))))
      (resume matcher-cor '()))))
```

现在过程 `same-fringe?` 可以这样写：

```
(define same-fringe?
  (lambda (tree1 tree2)
    (letrec ((tree-cor-1
              (make-leaf-gen-coroutine
               tree1
               matcher-cor))
             (tree-cor-2
              (make-leaf-gen-coroutine
               tree2
               matcher-cor))
             (matcher-cor
              (make-matcher-coroutine
               tree-cor-1
               tree-cor-2)))
      (matcher-cor 'start-ball-rolling))))
```

不幸的是Scheme的 `letrec` 语句如果想解析它引入的词法变量的相互递归调用，必须得把这个引用包围在一个 `lambda` 里。所以我们得这么写：

```
(define same-fringe?
  (lambda (tree1 tree2)
    (letrec ((tree-cor-1
              (make-leaf-gen-coroutine
               tree1
               (lambda (v) (matcher-cor v))))
             (tree-cor-2
              (make-leaf-gen-coroutine
               tree2
               (lambda (v) (matcher-cor v))))
             (matcher-cor
              (make-matcher-coroutine
               (lambda (v) (tree-cor-1 v))
               (lambda (v) (tree-cor-2 v)))))
      (matcher-cor 'start-ball-rolling))))
```

注意在这个版本的 `same-fringe` 里完全没有调用 `call/cc` 的痕迹。宏 `coroutine` 帮助我们处理了所有的协程。

---

[1]: 如果你的Scheme没有 `call/cc` 这个缩写，那么在你的初始化代码里加入 `(define call/cc call-with-current-continuation)` ，这样可以减少敲击键盘造成的手部劳损：）

第十四章 不确定性

麦卡锡的非确定运算符 `amb` 几乎和Lisp一样古老，尽管现在它已经从Lisp中消失了。 `amb` 接受一个或多个表达式，并在它们中进行一次“非确定”（或者叫“模糊”）选择，这个选择会让程序趋向于有意义。现在我们来探索一下Scheme内置的 `amb` 过程，该过程会对模糊的选项进行深度优先选择，并使用Scheme的控制操作符 `call/cc` 来回溯其他的选项。结果是一个优雅的回溯机制，该机制可用于在Scheme中对问题空间进行搜索而不需要另一种扩展了的语言。这种内嵌的恢复续延的机制可以用来实现Prolog风格的逻辑语言，但是更方便（sparer），因为这个操作符更像是Scheme的一个布尔运算符，使用时不需要特殊的上下文（context），而且也不依赖语言学的一些基础元素如逻辑变量和归纳法（unification）。

14.1 对amb的描述

最早的Scheme的教程SICP对 `amb` 进行了易于理解的描述，同时还给出了许多例子。说得直白一些， `amb` 接受零个或更多表达式并“不确定”的返回其中“一个”的值。因此：

```
(amb 1 2)
```

的结果可能为1或2。

不带参数调用 `amb` 则不会有返回值，而且应该会出错。因此：

```
(amb)
-->ERROR!!! amb tree exhausted
```

（我们后面再讨论这个错误信息。）

特别的，如果它的至少一个外层表达式收敛（converges）此时需要 `amb` 返回一个值，那么就不会出错，因此：

```
(amb 1 (amb))
```

而且：

都返回 `1` 。

很明显， `amb` 不能简单的等同于它的第一个子表达式，因为它必须返回一个“非错误”的值，如果有这种可能的话。然而，仅仅这样还不够：为使程序收敛的选择比单纯选择 `amb` 的子表达式要更加严格。 `amb` 应该返回让“整个”程序收敛的值。在这

个意义上，`amb` 是一个“神”一般的运算符。

比如：

```
(amb #f #t)
```

可以返回 `#f` 或 `#t` ，但是在程序：

```
(if (amb #f #t)
    1
    (amb))
```

中，第一个 `amb` 表达式必须返回 `#t` ，如果返回 `#f` ，那就会执行 `else` 分支，这会导致整个程序挂掉。

14.2 用Scheme实现amb

在我们的 `amb` 实现中，我们令 `amb` 的子表达式从左向右。也就是说，我们先选择第一个子表达式，如果不论怎样它都失败，那再选择第二个，如此等等。在回溯到前一个 `amb` 之前，程序控制流中后面出现的 `amb` 也被搜索以查看所有的可能性。换句话说，我们对 `amb` 的选择树进行了一个深度优先搜索，当我们碰到失败的情况时，我们就回溯到最近的节点来尝试其他的选择。（这叫做按时间顺序的回溯。）

我们首先定义一个机制来处理基本的错误的续延：

```
(define amb-fail '*)

(define initialize-amb-fail
  (lambda ()
    (set! amb-fail
      (lambda ()
        (error "amb tree exhausted")))))

(initialize-amb-fail)
```

当 `amb` 出错时，它调用绑定到 `amb-fail` 的续延。这个续延是在所有 `amb` 的选择树都被尝试过并且失败的情况下调用的。

我们把 `amb` 定义为一个宏，接受任意数量的参数。

```
(define-macro amb
  (lambda alts...
    `(let ((+prev-amb-fail amb-fail))
       (call/cc
        (lambda (+sk)

          ,@(map (lambda (alt)
                   `(call/cc
                     (lambda (+fk)
                       (set! amb-fail
                         (lambda ()
                           (set! amb-fail +prev-amb-fail)
                           (+fk 'fail)))
                       (+sk ,alt))))
                 alts...)

          (+prev-amb-fail))))))
```

对 `amb` 的调用被首先存储到 `+prev-amb-fail` 中， `amb-fail` 的值是此时的入口。这是因为 `amb-fail` 变量会被随着对可能选项的遍历被设置为不同的失败续延。

我们然后捕获 `amb` 的入口续延 `+sk` ，这样当求出一个“非失败”的值时，它可以马上退出 `amb` 。

每个序列中的选择 `alt` 都被尝试（Scheme中隐式的 `begin` 序列）。

首先，我们捕获当前续延 `+fk` ，把它包在一个过程中并把该过程赋给 `amb-fail` 。接着替换物被求值 `(+sk alt)` 。如果 `alt` 的求值没有失败，那么把它的返回值作为参数给续延 `+sk` ，这样马上就退出了 `amb` 的调用。如果 `alt` 失败了，就调用 `amb-fail` 。 `amb-fail` 做的第一件事是重新设置 `amb-fail` 为之前入口时的值。它接下来调用失败续延 `+fk` ，这个续延会尝试下个可能的选择（如果存在的话）。

如果所有选择都失败了， `amb` 入口的 `amb-fail` （我们之前把它存放在 `+prev-amb-fail` 中）会被调用。

14.3 在Scheme中使用amb

选择一个1到10之间的数字，我们可以这样写：

```
(amb 1 2 3 4 5 6 7 8 9 10)
```

毫无疑问这个程序会返回1（根据我们之前实现的策略），但这个与它的上下文有关，它完全可能返回给定的任何数字。

过程`number-between`是一种生成给定`lo`到`hi`（包括`lo`和`hi`在内）之间数字的抽象方法：

```
(define number-between
  (lambda (lo hi)
    (let loop ((i lo))
      (if (> i hi) (amb)
          (amb i (loop (+ i 1)))))))
```

因此`(number-between 1 6)`会首先生成1。如果失败了，继续循环，生成2。如果还是失败，我们就得到3，这样一直到6。6以后，`loop`以参数7被调用，这比6要大，调用`(amb)`。这会产生一个最终的错误（回忆之前我们所说的，单独的`(amb)`肯定会出现错误）这时，这个包含`(number-between 1 6)`的程序会按时间顺序依次回溯之前的`amb`调用，用另一种方式来满足这个调用。

`(amb)`一定失败的特点可以用于程序的 断言 中。

```
(define assert
  (lambda (pred)
    (if (not pred) (amb))))
```

调用`(assert pred)`确保了`pred`为真，否则它会让当前的`amb`选择点失败。

下面的程序用`assert`来生成一个小于等于其参数`hi`的素数：

```
(define gen-prime
  (lambda (hi)
    (let ((i (number-between 2 hi)))
      (assert (prime? i))
      i)))
```

这看起来也太简单了，只是当不论以任何数字（如20）调用这个过程，它永远会给出第一个解：2。

我们当然希望得到所有的解，而不是只有第一个。这种情况下，我们会希望得到所有比20小的素数。一种方法是在该过程输出了第一个解后，显式地调用失败续延。因此：

```
(amb)
=> 3
```

这样又会产生另一个失败续延，我们还可以继续调用它来得到另一个解。

```
(amb)
=> 5
```

这种方式的问题是程序首先在Scheme的命令提示符后面被调用，并且在Scheme的命令行上调用 `(amb)` 也可以得到成功的解。实际上，我们正在使用不同的程序（我们无法预计到底有多少！），并把信息从前一个传递到下一个。相反的，我们希望可以在任意上下文中调用某种形式然后返回这些解。为此我们定义了 `bag-of` 宏，该宏返回其参数的所有成功实例。（如果参数永远不能成功，就返回空列表）因此我们可以这样写：

```
(bag-of
  (gen-prime 20))
```

这样会返回：

```
(2 3 5 7 11 13 17 19)
```

宏 `bag-of` 定义如下：

```
(define-macro bag-of
  (lambda (e)
    `(let ((+prev-amb-fail amb-fail)
           (+results '()))
       (if (call/cc
            (lambda (+k)
              (set! amb-fail (lambda () (+k #f)))
              (let ((+v ,e))
                (set! +results (cons +v +results))
                (+k #t))))
           (amb-fail))
       (set! amb-fail +prev-amb-fail)
       (reverse! +results))))
```

`bag-of` 首先保存它的入口到 `amb-fail` 。它重新定义了 `amb-fail` 为一个在 `if` 测试中创建的本地续延。在这个测试中， `bag-of` 的参数 `e` 被求值，如果成功，它的结果被收集到一个叫 `+results` 的列表，并且以 `#t` 为参数调用本地续延。这会让 `if` 测试成功，导致 `e` 会在它的下一个回溯点被重新尝试。 `e` 的其他结果也通过这种方法获得并放进 `+results` 里。

最后，当 `e` 失败时，它会调用基本的 `amb-fail` ，即以 `#f` 为参数调用本地续延。这就把控制从 `if` 中转移出来。我们把 `amb-fail` 恢复到它上一个入口的值，并返回 `+results` 。（过程 `reverse!` 只是用来把结果以他们生成的顺序展现出来）

14.4 逻辑谜题

在解决逻辑谜题时，这种深度优先搜索与回溯相结合的方法的强大才能明显体现出来。这些问题用过程式的方式非常难以解决，但是可以用 `amb` 简洁、直截了当的解决，而且不会减少解决问题的魅力。

14.4.1 Kalotan谜题

Kalotan是一个奇特的部落。这个部落里所有男人都总是讲真话。所有的女人从来不会连续2句讲真话，也不会连续2句都讲假话。

一个哲学家（Worf）开始研究这些人。Worf不懂Kalotan的语言。一天他碰到一对Kalotan夫妻和他们的孩子Kibi。Worf问Kibi：“你是男孩吗？”Kibi用Kalotan语回答，Worf没听懂。

Wrof又问孩子的父母（他们都会说英语），其中一个人说：“Kibi说：‘我是个男孩。’”，另外一个人说：“Kibi是个女孩，Kibi撒谎了”。

请问这三个Kalotan人的性别。

解决的方法包括引进一堆变量，给它们赋上各种可能的值，把所有情况列举为一系列 `assert` 表达式。

变量： `parent1` , `parent2` , `kibi` 分别是父母（按照说话的顺序）和Kibi的性别。 `kibi-self-desc` 是Kibi用Kalotan语说的自己的性别。 `kibi-lied?` 表示Kibi是否说谎。

```
(define solve-kalotan-puzzle
  (lambda ()
    (let ((parent1 (amb 'm 'f))
          (parent2 (amb 'm 'f))
          (kibi (amb 'm 'f))
          (kibi-self-desc (amb 'm 'f))
          (kibi-lied? (amb #t #f)))
      (assert
       (distinct? (list parent1 parent2)))
      (assert
       (if (eqv? kibi 'm)
           (not kibi-lied?)))
      (assert
       (if kibi-lied?
           (xor
            (and (eqv? kibi-self-desc 'm)
                 (eqv? kibi 'f))
            (and (eqv? kibi-self-desc 'f)
                 (eqv? kibi 'm)))))
      (assert
       (if (not kibi-lied?)
           (xor
            (and (eqv? kibi-self-desc 'm)
                 (eqv? kibi 'm))
            (and (eqv? kibi-self-desc 'f)
                 (eqv? kibi 'f)))))
      (assert
       (if (eqv? parent1 'm)
           (and
            (eqv? kibi-self-desc 'm)
            (xor
             (and (eqv? kibi 'f)
                  (eqv? kibi-lied? #f))
             (and (eqv? kibi 'm)
                  (eqv? kibi-lied? #t))))))
      (assert
       (if (eqv? parent1 'f)
           (and
            (eqv? kibi 'f)
            (eqv? kibi-lied? #t))))
      (list parent1 parent2 kibi))))
```

对于辅助过程的一些说明：`distinct?` 过程返回 `true` ，如果其参数列表里所有参数都是不同的，否则返回 `false` 。过程 `xor` 只有当它的两个参数一个真一个假时才返回 `true` ，否则返回 `false` 。

输入 `(solve-kalotan-puzzle)` 会解决这个谜题。

14.4.2 地图着色

人们很早以前就知道（但知道1976年才证明）至少用四种颜色就可以给地球的地图着色，也就是说给所有国家着色并保证相邻的国家的颜色是不同的。为了验证确实是这样的，我们编写下面的程序，并指出非确定性编程是如何为之提供便利的。

下面的这段程序解决了西欧的地图着色问题。这个问题和其用Prolog语言的解法在《the Art of Prolog》中给出。（如果你能比较我们与那本书里的解法应该很有益处）

过程 `choose-color` 非确定的返回四种颜色之一：

```
(define choose-color
  (lambda ()
    (amb 'red 'yellow 'blue 'white)))
```

在我们的解法中，我们为每个国家建立了一个数据结构。该结构是一个三元素的列表：第一个元素表示国家名，第二个元素是颜色，第三个元素是它相邻国家的颜色。注意我们用国家的首字母作为颜色的变量，即比利时（Belgium）的列表是 `(list 'belgium b (list f h l g))` ，因为——按照这个问题列表——比利时的邻国是法国(France)，荷兰(Holland)，卢森堡(Luxembourg)，德国(Germany)。

一旦我们给每个国家创建了列表，我们 仅仅 需要陈述他们应该满足的条件，即每个国家不能与邻国有相同的颜色。换句话说，对每个国家的列表，第二个元素的值应该不在第三个元素（列表）中。

```
(define color-europe
  (lambda ()

    ;choose colors for each country
    (let ((p (choose-color)) ;Portugal
          (e (choose-color)) ;Spain
          (f (choose-color)) ;France
          (b (choose-color)) ;Belgium
          (h (choose-color)) ;Holland
          (g (choose-color)) ;Germany
          (l (choose-color)) ;Luxemb
          (i (choose-color)) ;Italy
          (s (choose-color)) ;Switz
          (a (choose-color)) ;Austria
          )

      ;construct the adjacency list for
      ;each country: the 1st element is
      ;the name of the country; the 2nd
      ;element is its color; the 3rd
      ;element is the list of its
      ;neighbors' colors
      (let ((portugal
             (list 'portugal p
                   (list e)))
```

```
              (spain
               (list 'spain e
                     (list f p)))
              (france
               (list 'france f
                     (list e i s b g l)))
              (belgium
               (list 'belgium b
                     (list f h l g)))
              (holland
               (list 'holland h
                     (list b g)))
              (germany
               (list 'germany g
                     (list f a s h b l)))
              (luxembourg
               (list 'luxembourg l
                     (list f b g)))
              (italy
               (list 'italy i
                     (list f a s)))
              (switzerland
               (list 'switzerland s
                     (list f i a g)))
              (austria
               (list 'austria a
                     (list i s g))))
          (let ((countries
                 (list portugal spain
                       france belgium
                       holland germany
                       luxembourg
                       italy switzerland
                       austria)))

            ;the color of a country
            ;should not be the color of
            ;any of its neighbors
            (for-each
             (lambda (c)
               (assert
                (not (memq (cadr c)
                           (caddr c)))))
             countries)

            ;output the color
            ;assignment
            (for-each
             (lambda (c)
               (display (car c))
               (display " ")
               (display (cadr c))
               (newline))
```

```
            countries))))))
```

输入`(color-europe)`来得到一个颜色-国家对应表。

---

1. SICP把这个过程命名为`require`。我们使用`assert`标识符是为了避免与用来从其他文件中加载代码的`require`标识符混淆。

第十五章 引擎

引擎表示服从时间抢占的运算过程。换句话说，一个引擎下面的运算过程是普通的程序作为定时器可抢占的进程。

一个引擎用三个参数来调用：

1. 分配时间片（运行时间单元）的数目
2. 成功过程
3. 失败过程

如果引擎的计算在分配的时间片内完成了，那么就把计算的结果作为参数来调用成功过程，如果没有计算完成，那么把未计算完的部分作为参数来调用失败过程。

比如，考虑一个引擎，其下的运算是一个循环，该循环打印非负整数的序列。该引擎用下面的 `make-engine` 过程（后面会定义该过程）创建，`make-engine` 接受一个程序（即该引擎下面的计算过程）为参数，并返回对应的引擎。

```
(define printn-engine
  (make-engine
    (lambda ()
      (let loop ((i 0))
        (display i)
        (display " ")
        (loop (+ i 1))))))
```

下面调用 `pritn-engine` ：

```
(define *more* #f)
(printn-engine 50 list (lambda (ne) (set! *more* ne)))
=>  0 1 2 3 4 5 6 7 8 9
```

也就是循环打印到某个特定的数（这里是 `9` ）然后就失败(fail)了，因为时钟中断了。然而，我们定义的失败过程把fail掉的引擎赋值给了全局变量 `*more*` 。这样我们就可以从上个引擎中断的地方恢复：

```
(*more* 50 list (lambda (ne) (set! *more* ne)))
=>  10 11 12 13 14 15 16 17 18 19
```

我们现在来构建引擎，使用 `call/cc` 来捕获一个失败引擎未完成的计算。首先我们会构造一个flat引擎，也就是说该引擎的计算中不能运行其他引擎。稍后我们会让代码更通用，实现nestable引擎，这样的引擎可以调用其他引擎。但是不管那种引擎，我们都需要一个定时的东西，时钟（clock）。

15.1 时钟

我们的引擎假设有一个全局的时钟或可中断的定时器来记录程序运行的时间片。我们假设下面的时钟接口——你通常应该很容易把Scheme提供的时钟接口（如果有的话）打包成下面这种类型。（附录D用Scheme的Guile方言定义了一个时钟）

我们的 `clock` 过程的内部状态包括以下两项：

1. 剩余的时间片的数目，以及
2. 一个中断处理器(handler)，当时钟的时间片用完了的时候被调用。

`clock` 允许下面的操作：

1. `(clock 'set-handler h)` 设置中断处理器为 `h` 。
2. `(clock 'set n)` 把时钟的剩余时间片重置为 `n` ，返回之前的值。

`n` 的取值范围是所有非负整数以及一个叫 `*infinity*` 的原子。一个时钟如果有 `*infinity*` 的时间片永远不会终止，所以也用不着设置中断处理器。这样的时钟总是“静止的”或“停止的”（定时器的工作就是：减到0并中断，而这样的时钟永远不会减到0并中断，所以处于“非工作”状态）。让一个时钟停止只要把时间片设为 `*infinity*` 即可。

时钟的处理器被设置为一个程序，比如：

```
(clock 'set-handler
  (lambda ()
    (error "Say goodnight, cat!")))

(clock 'set 9)
```

这样 `9` 个时间片过去后会产生一个错误，并且显示的错误信息是 `"Say goodnight, cat!"`

15.2 flat引擎

我们将首先设置时钟的中断处理器。注意这个处理器只有在“工作”状态的时钟用完时间片后才会被调用。这只有引擎计算失败时才发生，因为只有引擎设置时钟。

处理器捕获当前的续延，也就是当前失败引擎的剩余计算部分。这个续延被保存到全局的 `*engine-escape*` 变量中。该变量存放当前引擎的续延。因此时钟处理器捕获失败引擎的剩余部分并把它发送到引擎代码的出口。这样所需的失败处理才能执行。

```
(define *engine-escape* #f)
(define *engine-entrance* #f)

(clock 'set-handler
  (lambda ()
    (call/cc *engine-escape*)))
```

让我们来看一下引擎代码的内部。如上所述，`make-engine` 接受一个程序并为之构造一个引擎：

```
(define make-engine
  (lambda (th)
    (lambda (ticks success failure)
      (let* ((ticks-left 0)
             (engine-succeeded? #f)
             (result
              (call/cc
               (lambda (k)
                 (set! *engine-escape* k)
                 (let ((result
                        (call/cc
                         (lambda (k)
                           (set! *engine-entrance* k)
                           (clock 'set ticks)
                           (let ((v (th)))
                             (*engine-entrance* v))))))
                   (set! ticks-left (clock 'set *infinity*))
                   (set! engine-succeeded? #t)
                   result)))))
        (if engine-succeeded?
            (success result ticks-left)
            (failure
             (make-engine
              (lambda ()
                (result 'resume)))))))))
```

首先我们引入变量 `ticks-left` 和 `engine-succeeded?` 。前者保存引擎的程序应该在多少时间片内完成。后者是一个标志，表示引擎是否成功。

我们接下来在两层对 `call/cc` 的调用中执行引擎的程序。第一个 `call/cc` 捕获的续延被失败引擎用来退出其引擎的计算。这个续延被保存到全局的 `*engine-escape*` 变量中。第二个 `call/cc` 捕获一个内部的续延，该续延会被 `th` 的返回值使用，如果 `th` 完成了的话。这个续延保存在全局的 `*engine-entrance*` 变量中。

查看上面的代码，我们能发现在捕获续延 `*engine-escape*` 和 `*engine-entrance*` 后，我们设置时钟的时间片为允许允许的时间并运行 `th` 。如果 `th` 成功了，其返回回值 `v` 被发送到续

延 `*engine-escape*` ，然后时钟就停止了，剩下的时间片的数量就确定了，并且标记 `engine-succeeded?` 被设置为真。我们现在略过(?) `*engine-escape*` 续延，并执行最后一段选择语句：由于我们知道引擎成功了，我们以执行结果和剩余的时间片为参数调用 `success` 过程。

如果程序 `th` 没能在制定时间完成，就会被中断。这会调用时钟的中断处理器，来捕获当前失败程序的续延，并把它传给 `*engine-escape*` 变量。这样就把 `result` 变量设置为失败任务的续延，我们现在执行最后一个 `if` 语句，由于 `engine-succeeded?` 是 `false` ，我们以 `result` 构造一个新引擎并作为参数调用 `failure` 过程。

注意当一个失败的引擎被移除时，it will traverse the control path charted by the first run of the original engine.尽管如此，因为我们总是显式的使用保存在 `*engine-entrance*` 和 `*engine-escape*` 变量中的续延，而且我们总是在开启引擎计算前重新设置它们，我们能保证跳转总是会到当前执行的引擎代码。

15.3 交叠(nestable)的引擎

为了让上面的代码更通用化，适应交叠的引擎，我们需要引入一些时间片管理的机制，来为交叠运行的所有引擎管理正确的时间片。

为了跑一个新引擎（子引擎），我们需要停止当前引擎（父引擎）。然后需要给子引擎分配适当的时间。这可不是直接在程序代码里设置那样，因为给子引擎分配比父引擎剩下的时间片更多的时间片是不对的。在子引擎跑完后我们还得更新父引擎剩余的时间片。如果子引擎在给定时间内跑完，所有它剩下的时间片都还给父引擎。如果子引擎要求的时间片被拒绝（因为父引擎的剩余时间片都无法满足），那么如果子引擎失败了，父引擎也会失败，但是必须记得在重启父引擎时也重启子引擎，其时间片仍然是之前需要的那些。

我们需要用 `fluid-let` 来声明全局的 `*engine-escape*` 和 `*engine-entrance*` 变量，因为每个引擎都必须有它自己的这两个做控制用的续延。当引擎退出时（不论是成功还是失败）， `fluid-let` 会保证其外层引擎会接管这个控制(sentinel)。

考虑到以上这些，可交叠引擎的代码应该像下面这样：

```scheme
(define make-engine
  (lambda (th)
    (lambda (ticks s f)
      (let* ((parent-ticks
               (clock 'set *infinity*))

              ;A child can't have more ticks than its parent's
              ;remaining ticks
              (child-available-ticks
               (clock-min parent-ticks ticks))

              ;A child's ticks must be counted against the parent
              ;too
```

```
               (parent-ticks-left
                (clock-minus parent-ticks child-available-ticks))

               ;If child was promised more ticks than parent could
               ;afford, remember how much it was short-changed by
               (child-ticks-left
                (clock-minus ticks child-available-ticks))

               ;Used below to store ticks left in clock
               ;if child completes in time
               (ticks-left 0)

               (engine-succeeded? #f)

               (result
                (fluid-let ((*engine-escape* #f)
                            (*engine-entrance* #f))
                  (call/cc
                   (lambda (k)
                     (set! *engine-escape* k)
                     (let ((result
                            (call/cc
                             (lambda (k)
                               (set! *engine-entrance* k)
                               (clock 'set child-available-ticks)

                               (let ((v (th)))

                                 (*engine-entrance* v))))))
                       (set! ticks-left
                         (let ((n (clock 'set *infinity*)))
                           (if (eqv? n *infinity*) 0 n)))
                       (set! engine-succeeded? #t)
                       result))))))

          ;Parent can reclaim ticks that child didn't need
          (set! parent-ticks-left
            (clock-plus parent-ticks-left ticks-left))

          ;This is the true ticks that child has left --
          ;we include the ticks it was short-changed by
          (set! ticks-left
            (clock-plus child-ticks-left ticks-left))

          ;Restart parent with its remaining ticks
          (clock 'set parent-ticks-left)
          ;The rest is now parent computation

          (cond
           ;Child finished in time -- celebrate its success
           (engine-succeeded? (s result ticks-left))

           ;Child failed because it ran out of promised time --
```

```
            ;call failure procedure
            ((= ticks-left 0)
             (f (make-engine (lambda () (result 'resume)))))

            ;Child failed because parent didn't have enough time,
            ;ie, parent failed too.  If so, when parent is
            ;resumed, its first order of duty is to resume the
            ;child with its fair amount of ticks
            (else
             ((make-engine (lambda () (result 'resume)))
              ticks-left s f)))))))
```

注意我们使用算术运算符`clock-min`，`clock-minus`和`clock-plus`替代了`min`，`-`和`+`。这是因为时钟算术的值除了整数之外还包含`*infinity*`。有些Scheme的方言在它们的算术运算中提供了`*infinity*`的值——如果这样的话，你就可以用这些通用的算术运算符了。如果没有的话，定义这几个增强运算符也是很简单的事情。

---

1. 在Guile中，你可以`(define *infinity* (/ 1 0))`。

第十六章 命令行脚本

如果能把我们想做的东西写到一个文件或脚本中，并且像执行其他操作系统命令一样执行的话通常会非常方便。一些重量级的程序通常以脚本的形式提供接口，用户可以经常编写他们自己的脚本或修改已有的脚本来满足特定的需求。毫无疑问大部分的编程任务都以脚本的形式来执行。对于很多用户而言，这是他们唯一会做的编程了。

Unix或DOS等操作系统（以及Windows系统提供的命令行接口）都提供了脚本的机制。但是这些脚本语言都相当的不成熟。通常一个脚本就是一串可以在命令行上输入的命令。这样用户可以免于每次用这些命令（或相似的命令）都重新输入。某些脚本语言包含一些简单的编程功能如条件语句和循环，但这就是所有的了。这对于简单的程序是足够了。但是当脚本越来越大，要求越来越高——大部分情况都是如此——人们通常会觉得需要一些功能全面的编程语言。包含足够操作系统接口的Scheme语言让脚本编写变得简单而可维护。

这一节会描述如何在Scheme中编写脚本。由于Scheme有太多方言一节太多实现不同的实现方法，我们专注于使用MzScheme方言，附录A中讲解了如果用其他方言需要有哪些修改。我们现在也专门讲解Unix操作系统。附录B里讨论了DOS系统需要注意的问题。

16.1 再来一次Hello，World！

我们现在来创建一个Scheme脚本来对世界说hello。这对于通常的脚本语言也算不上什么难事。然而，为了后期上道编写更复杂的脚本，我们必须理解如何用Scheme来编写这个HelloWorld。首先，一个通常的Unix版的HelloWorld是一个文件，里面的内容如下：

```
echo Hello, World!
```

这里使用了命令`echo`，这个脚本可以被命名为`hello`，使用下面的命令使之可执行：

```
chmod +x hello
```

然后把它放在`PATH`环境变量中的任意一个目录下。然后任何时候从命令行输入

```
hello
```

就会输出上面的问候。

Scheme的hello脚本也会用Scheme产生相同的输出（见下面的脚本），但是我们得做点什么，让操作系统知道它应该用Scheme来分析文件中的命令，而不是用它默认的脚本语言。Scheme的脚本文件，有命名为hello，内容如下：

```
":"; exec mzscheme -r $0 "$@"

(display "Hello, World!")
(newline))
```

除了第一行以外都是Scheme代码。然而第一行就是把这些代码指定为“脚本”的神奇之处。当用户在Unix命令行上输入 `hello` 的时候，Unix会像读取一般的脚本一样来读取这个文件。首先读到一个 `":"` ，这是一个shell的空语句。后面的;是分隔符。下一个命令是 `exec` 。 `exec` 告诉Unix放弃当前脚本的执行并转而执行 `mzscheme -r $0 "$@"` ，这里参数 `$0` 会被替换为当前文件的名字，参数 `$@` 会被替换为用户运行该脚本时附加的参数。（在本例中没有参数）

我们现在事实上以及把 `hello` 命令变换为另一个不同的命令，即：

```
mzscheme -r /whereveritis/hello
```

其中 `/whereveritis/hello` 是 `hello` 文件的路径名。

mzscheme命令调用了MzScheme的可执行文件。 `-r` 选项告诉它把紧跟在该选项后面的参数作为一个Scheme文件来加载，在这之前还要把所有其他参数（如果有的话）放进一个叫 `argv` 的向量中（在本例中， `argv` 是一个空向量）。

因此，Scheme脚本会作为一个Scheme文件来执行，而且该文件中的Scheme代码还可以通过 `argv` 访问到所有该脚本原先的参数。

现在，Scheme不得不来处理这个脚本中的第一行了。正如我们所看到的，这一行可是一个精心构造的Shell脚本。 `":"` 是一个Scheme中自求值的字符串所以没有关系。 `;` 则开启了Scheme的注释，因此后面的exec等代码都被安全的忽略掉了。文件剩下的部分都是Scheme代码，被按顺序求值，所有的求值完成后，Scheme就退出了。

总之，在命令提示符后面输入 `hello` 会产生：

```
Hello, World!
```

并把命令提示符返回给你。

16.2 带参数的脚本

Scheme脚本使用argv变量来引用它的参数。例如，下面的脚本输出其所有参数，每个一行：

```
":"; exec mzscheme -r $0 "$@"

;Put in argv-count the number of arguments supplied

(define argv-count (vector-length argv))

(let loop ((i 0))
  (unless (>= i argv-count)
    (display (vector-ref argv i))
    (newline)
    (loop (+ i 1))))
```

我们把这个脚本命名为 `echoall` 。调用 `echoall 1 2 3` 会显示：

```
1
2
3
```

注意脚本名称不包括在参数向量中。

16.3 例子

我们现在来解决一些更大的问题。我们需要在两台电脑之间传输文件，而唯一的方式是使用一张3.6英寸的软盘作为媒介。我们需要一个 `split4floppy` 的脚本来把大于1.44MB的文件分割为软盘能装下的小块。脚本 `split4floppy` 如下：

```
":";exec mzscheme -r $0 "$@"

;floppy-size = number of bytes that will comfortably fit on a
;              3.5" floppy

(define floppy-size 1440000)

;split splits the bigfile f into the smaller, floppy-sized
;subfiles, viz, subfile-prefix.1, subfile-prefix.2, etc.

(define split
  (lambda (f subfile-prefix)
    (call-with-input-file f
      (lambda (i)
        (let loop ((n 1))
          (if (copy-to-floppy-sized-subfile i subfile-prefix n)
            (loop (+ n 1))))))))

;copy-to-floppy-sized-subfile copies the next 1.44 million
;bytes (if there are less than that many bytes left, it
;copies all of them) from the big file to the nth
```

```
;subfile.  Returns true if there are bytes left over,
;otherwise returns false.

(define copy-to-floppy-sized-subfile
  (lambda (i subfile-prefix n)
    (let ((nth-subfile (string-append subfile-prefix "."
                                      (number->string n))))
      (if (file-exists? nth-subfile) (delete-file nth-subfile))
      (call-with-output-file nth-subfile
        (lambda (o)
          (let loop ((k 1))
            (let ((c (read-char i)))
              (cond ((eof-object? c) #f)
                    (else
                     (write-char c o)
                     (if (< k floppy-size)
                         (loop (+ k 1))
                         #t))))))))))

;bigfile = script's first arg
;        = the file that needs splitting

(define bigfile (vector-ref argv 0))

;subfile-prefix = script's second arg
;               = the basename of the subfiles

(define subfile-prefix (vector-ref argv 1))

;Call split, making subfile-prefix.{1,2,3,...} from
;bigfile

(split bigfile subfile-prefix)
```

脚本 `split4floppy` 用如下方法调用：

```
split4floppy largefile chunk
```

这会把 `largefile` 分割成 `chunk.1` 、 `chunk.2` 等等，每个小块文件都能装进软盘中。

所有 `chunk.i` 都移动到目标电脑上以后可以通过把 `chunk.i` 按顺序拼起来还原 `largefile` 原文件，在Unix上这样做：

```
cat chunk.1 chunk.2 ... > largefile
```

在DOS下这样做：

```
copy /b chunk.1+chunk.2+... largefile
```

第十七章 CGI脚本

（警告：缺乏适当安全防护措施的CGI脚本可能会让您的网站陷入危险状态。本文中的脚本只是简单的样例而不保证在真实网站使用是安全的。）
CGI脚本是驻留在Web服务器上的脚本，而且可以被客户端（浏览器）运行。客户端通过脚本的URL来访问脚本，就像访问普通页面一样。服务器识别出请求的URL是一个脚本，于是就运行该脚本。服务器如何识别特定的URL为脚本取决于服务器的管理员。在本文中我们假设脚本都存放在一个单独的文件夹，名为cgi-bin。因此，www.foo.org网站上的 `testcgi.scm` 脚本可以通过 http://www.foo.org/cgi-bin/testcgi.scm 来访问。

服务器以 `nobody` 用户的身份来运行脚本，不应当期望这个用户有 `PATH` 的环境变量或者该变量正确设置（这太主观了）。因此用Scheme编写的脚本的“引导行”会比我们在一般Scheme脚本中更加清楚才行。也就是说，下面这行代码：

```
":";exec mzscheme -r $0 "$@"
```

隐式的假设有一个特定的shell（如bash），而且设置好了 `PATH` 变量，而mzscheme程序在PATH的路径里。对于CGI脚本，我们需要多写一些：

```
#!/bin/sh
":";exec /usr/local/bin/mzscheme -r $0 "$@"
```

这样指定了shell和Scheme可执行文件的绝对路径。控制从shell交接给Scheme的过程和普通脚本一致。

17.1 例：显示环境变量

下面是一个Scheme编写的CGI脚本的示例， `testcgi.scm` 。该文件会输出一些常用CGI环境变量的设置。这些信息作为一个新的，刚刚创建的页面返回给浏览器。返回的页面就是该CGI脚本向标准输出里写入的任何东西。这就是CGI脚本如何回应对它们的调用——通过返回给它们（客户端）一个新页面。

注意脚本首先输出下面这行：

```
content-type: text/plain
```

后面跟一个空行。这是Web服务器提供页面服务的标准方式。这两行不会在页面上显示出来。它们只是提醒浏览器下面将发送的页面是纯文本（也就是非标记）文字。这样浏览器就会恰当的显示这个页面了。如果我们要发送的页面是用HTML标记的， `content-type` 就是 `text/html` 。

下面是脚本 `testcgi.scm` ：

```
#!/bin/sh
":";exec /usr/local/bin/mzscheme -r $0 "$@"

;Identify content-type as plain text.

(display "content-type: text/plain") (newline)
(newline)

;Generate a page with the requested info.  This is
;done by simply writing to standard output.

(for-each
 (lambda (env-var)
   (display env-var)
   (display " = ")
   (display (or (getenv env-var) ""))
   (newline))
 '("AUTH_TYPE"
   "CONTENT_LENGTH"
   "CONTENT_TYPE"
   "DOCUMENT_ROOT"
   "GATEWAY_INTERFACE"
   "HTTP_ACCEPT"
   "HTTP_REFERER" ; [sic]
   "HTTP_USER_AGENT"
   "PATH_INFO"
   "PATH_TRANSLATED"
   "QUERY_STRING"
   "REMOTE_ADDR"
   "REMOTE_HOST"
   "REMOTE_IDENT"
   "REMOTE_USER"
   "REQUEST_METHOD"
   "SCRIPT_NAME"
   "SERVER_NAME"
   "SERVER_PORT"
   "SERVER_PROTOCOL"
   "SERVER_SOFTWARE"))
```

`testcgi.scm` 可以直接从浏览器上打开，URL是：

http://www.foo.org/cgi-bin/testcgi.scm

此外， `testcgi.scm` 也可以放在HTML文件的链接中，这样可以直接点击，如：

```
... To view some common CGI environment variables, click
<a href="http://www.foo.org/cgi-bin/testcgi.scm">here</a>.
...
```

而一旦触发了`testcg.scm`，它就会生成一个包括环境变量设置的纯文本页面。下面是一个示例输出：

```
AUTH_TYPE =
CONTENT_LENGTH =
CONTENT_TYPE =
DOCUMENT_ROOT = /home/httpd/html
GATEWAY_INTERFACE = CGI/1.1
HTTP_ACCEPT = image/gif, image/x-xbitmap, image/jpeg, image/pjpeg,
HTTP_REFERER =
HTTP_USER_AGENT = Mozilla/3.01Gold (X11; I; Linux 2.0.32 i586)
PATH_INFO =
PATH_TRANSLATED =
QUERY_STRING =
REMOTE_HOST = 127.0.0.1
REMOTE_ADDR = 127.0.0.1
REMOTE_IDENT =
REMOTE_USER =
REQUEST_METHOD = GET
SCRIPT_NAME = /cgi-bin/testcgi.scm
SERVER_NAME = localhost.localdomain
SERVER_PORT = 80
SERVER_PROTOCOL = HTTP/1.0
SERVER_SOFTWARE = Apache/1.2.4
```

17.2 示例：显示选择的环境变量

`testcgi.scm`没有从用户获得任何输入。一个更专注的脚本会从用户那里获得一个环境变量，然后输出这个变量的设置，此外不返回任何东西。为了做这个，我们需要一个机制把参数传递给CGI脚本。HTML的表单提供了这种功能。下面是完成这个目标的一个简单的HTML页面：

```
<html>
<head>
<title>Form for checking environment variables</title>
</head>
<body>

<form method=get
      action="http://www.foo.org/cgi-bin/testcgi2.scm">
Enter environment variable: <input type=text name=envvar size=30>
<p>

<input type=submit>
</form>

</body>
</html>
```

用户在文本框中输入希望的环境变量（如 `GATEWAY_INTERFACE` ）并点击提交按钮。这会把所有表单里的信息——这里，参数 `envvar` 的值是 `GATEWAY_INTERFACE` ——收集并发送到该表单对应的CGI脚本即 `testcgi2.scm` 。这些信息可以用两种方法来发送：

1. 如果表单的 `method` 属性是 `GET` （默认），那么这些信息通过环境变量 `QUERY_STRING` 来传递给脚本
2. 如果表单的 `method` 属性是 `POST` ，那么这些信息会在稍后发送到CGI脚本的标准输入中。

我们的表单使用 `QUERY_STRING` 的方式。

把信息从 `QUERY_STRING` 中提取出来并输出相应的页面是 `testcgi2.scm` 脚本的事情。

发给CGI脚本的信息，不论通过环境变量还是通过标准输入，都被格式化为一串“参数/值”的键值对。键值对之间用 `&` 字符分隔开。每个键值对中参数的名字在前面而且与参数值之间用 `=` 分开。这种情况下，只有一个键值对，即 `envvar=GATEWAY_INTERFACE` 。

下面是 `testcgi2.scm` 脚本：

```
#!/bin/sh
":";exec /usr/local/bin/mzscheme -r $0 "$@"

(display "content-type: text/plain") (newline)
(newline)

;string-index returns the leftmost index in string s
;that has character c

(define string-index
  (lambda (s c)
    (let ((n (string-length s)))
      (let loop ((i 0))
        (cond ((>= i n) #f)
              ((char=? (string-ref s i) c) i)
              (else (loop (+ i 1))))))))

;split breaks string s into substrings separated by character c

(define split
  (lambda (c s)
    (let loop ((s s))
      (if (string=? s "") '()
          (let ((i (string-index s c)))
            (if i (cons (substring s 0 i)
                        (loop (substring s (+ i 1)
                                         (string-length s))))
                (list s))))))))

(define args
  (map (lambda (par-arg)
         (split #\= par-arg))
       (split #\& (getenv "QUERY_STRING"))))

(define envvar (cadr (assoc "envvar" args)))

(display envvar)
(display " = ")
(display (getenv envvar))

(newline)
```

注意辅助过程 `split` 把 `QUERY_STRING` 用 `&` 分隔为键值对并进一步用 `=` 把参数名和参数值分开。（如果我们是用 `POST` 方法，我们需要把参数名和参数值从标准输入中提取出来。）

`<input type=text>` 和 `<input type=submit>` 是HTML表单的两个不同的输入标签。参考文献27来查看全部。

17.3 CGI脚本相关问题（utilities）

在上面的例子中，参数名和参数值都假设没有包含 `=` 或 `&` 字符。通常情况他们会包含。为了适应这种字符，而不会不小心把他们当成分割符，CGI参数传递机制要求所有除了字母、数字和下划线以外的“特殊”字符都要编码进行传输。空格被编码为 `+` ，其他的特殊字符被编码为3个字符的序列，包括一个 `%` 字符紧跟着这个字符的16进制码。因此， `20% + 30% = 50%, &c.` 会被编码为：

```
20%25+%2b+30%25+%3d+50%25%2c+%26c%2e
```

（空格变成 `+` ； `%` 变为 `%25` ； `+` 变为 `%2b` ； `=` 变为 `%3d` ； `,` 变为 `%2c` ； `&` 变为 `%26` ； `.` 变为 `%2e` ）

除了把获得和解码表单的代码写在每个CGI脚本中，把这些函数放在一个库文件 `cgi.scm` 中。这样 `testcgi2.scm` 的代码写起来更紧凑：

```
#!/bin/sh
":";exec /usr/local/bin/mzscheme -r $0 "$@"

;Load the cgi utilities

(load-relatve "cgi.scm")

(display "content-type: text/plain") (newline)
(newline)

;Read the data input via the form

(parse-form-data)

;Get the envvar parameter

(define envvar (form-data-get/1 "envvar"))

;Display the value of the envvar

(display envvar)
(display " = ")
(display (getenv envvar))
(newline)
```

这个简短一些的CGI脚本用了两个定义在 `cgi.scm` 中的通用过程。 `parse-form-data` 过程读取用户通过表单提交的数据，包括参数和值。

`form-data-get/1` 找到与特定参数关联的值。

`cgi.scm` 定义了一个全局表叫 `*form-data-table*` 来存放表单数据。

```
;Load our table definitions

(load-relative "table.scm")

;Define the *form-data-table*

(define *form-data-table* (make-table 'equ string=?))
```

使用诸如 `parse-form-data` 等通用过程的一个好处是我们可以不用管用户是用那种方式（get或post）提交的数据。

```
(define parse-form-data
  (lambda ()
    ((if (string-ci=? (or (getenv "REQUEST_METHOD") "GET") "GET")
         parse-form-data-using-query-string
         parse-form-data-using-stdin))))
```

环境变量 `REQUEST_METHOD` 表示使用那种方式传送表单数据。如果方法是 `GET` ，那么表单数据被作为字符串通过另一个环境变量 `QUERY_STRING` 传输。辅助过程 `parse-form-data-using-query-string` 用来拆散 `QUERY_STRING` ：

```
(define parse-form-data-using-query-string
  (lambda ()
    (let ((query-string (or (getenv "QUERY_STRING") "")))
      (for-each
       (lambda (par=arg)
         (let ((par/arg (split #\= par=arg)))
           (let ((par (url-decode (car par/arg)))
                 (arg (url-decode (cadr par/arg))))
             (table-put!
              *form-data-table* par
              (cons arg
                    (table-get *form-data-table* par '()))))))
       (split #\& query-string)))))
```

辅助过程 `split` ，和它的辅助过程 `string-index` ，在第二节中定义过了。正如之前提到的，输入的表单数据是一串用 `&` 分割的键值对。每个键值对中先是参数名，然后是一个 `=` 号，后面是值。每个键值对都放到一个全局的表 `*form-data-table*` 里。

每个参数名和参数值都被编码了，所以我们需要用 `url-decode` 过程来解码得到它们的真实表示。

```
(define url-decode
  (lambda (s)
    (let ((s (string->list s)))
      (list->string
       (let loop ((s s))
         (if (null? s) '()
             (let ((a (car s)) (d (cdr s)))
               (case a
                 ((#\+) (cons #\space (loop d)))
                 ((#\%) (cons (hex->char (car d) (cadr d))
                              (loop (cddr d))))
                 (else (cons a (loop d)))))))))))
```

`+` 被转换为空格，通过过程 `hex->char`，`%xy` 这种形式的词也被转换为其ascii编码是十六进制数 `xy` 的字符。

```
(define hex->char
  (lambda (x y)
    (integer->char
     (string->number (string x y) 16))))
```

我们还需要一个处理POST方法传输数据的程序。辅助过程 `parse-form-data-using-stdin` 就是做这个的。

```
(define parse-form-data-using-stdin
  (lambda ()
    (let* ((content-length (getenv "CONTENT_LENGTH"))
           (content-length
             (if content-length
                 (string->number content-length) 0))
           (i 0))
      (let par-loop ((par '()))
        (let ((c (read-char)))
          (set! i (+ i 1))
          (if (or (> i content-length)
                  (eof-object? c) (char=? c #\=))
              (let arg-loop ((arg '()))
                (let ((c (read-char)))
                  (set! i (+ i 1))
                  (if (or (> i content-length)
                          (eof-object? c) (char=? c #\&))
                      (let ((par (url-decode
                                   (list->string
                                     (reverse! par))))
                            (arg (url-decode
                                   (list->string
                                     (reverse! arg)))))
                        (table-put! *form-data-table* par
                          (cons arg (table-get *form-data-table*
                                      par '())))
                        (unless (or (> i content-length)
                                    (eof-object? c))
                          (par-loop '())))
                      (arg-loop (cons c arg)))))
              (par-loop (cons c par))))))))
```

POST方法通过脚本的标准输入传输表单数据。传输的字符数放在环境变量 `CONTENT_LENGTH` 里。 `parse-form-data-using-stdin` 从标准输入读取对应的字符，也像之前那样设置 `*form-data-table*` ，保证参数名和值被解码。

剩下就是从 `*form-data-table*` 取回特定参数的值。主要这个这个表中每个参数都关联着一个列表，这是为了适应一个参数多个值的情况。 `form-data-get` 取回一个参数对应的所有值。如果只有一个值，就返回这个值。

```
(define form-data-get
  (lambda (k)
    (table-get *form-data-table* k '())))
```

`form-data-get/1` 返回一个参数的第一个（或最重要的）值。

```
(define form-data-get/1
  (lambda (k . default)
    (let ((vv (form-data-get k)))
      (cond ((pair? vv) (car vv))
            ((pair? default) (car default))
            (else "")))))
```

在我们目前的例子当中，CGI脚本都是生成纯文本，通常我们希望生成一个HTML页面。把CGI脚本和HTML表单结合起来生成一系列带有表单的HTML页面是很常见的。把不同方法的响应代码放在一个CGI脚本里也是很常见的。不论任何情况，有一些辅助过程把字符串输出为HTML格式（即，把HTML特殊字符进行编码））都是很有帮助的：

```
(define display-html
  (lambda (s . o)
    (let ((o (if (null? o) (current-output-port)
                 (car o))))
      (let ((n (string-length s)))
        (let loop ((i 0))
          (unless (>= i n)
            (let ((c (string-ref s i)))
              (display
               (case c
                 ((#\<) "&lt;")
                 ((#\>) "&gt;")
                 ((#\") "&quot;")
                 ((#\&) "&amp;")
                 (else c)) o)
              (loop (+ i 1)))))))))
```

17.4 一个CGI做的计算器

下面是一个CGI计算器的脚本，`cgicalc.scm`，使用了Scheme任意精度的算术功能。

```
#!/bin/sh
":";exec /usr/local/bin/mzscheme -r $0

;Load the CGI utilities
(load-relative "cgi.scm")

(define uhoh #f)

(define calc-eval
  (lambda (e)
    (if (pair? e)
        (apply (ensure-operator (car e))
```

```
                (map calc-eval (cdr e)))
          (ensure-number e))))

(define ensure-operator
  (lambda (e)
    (case e
      ((+) +)
      ((-) -)
      ((*) *)
      ((/) /)
      ((**) expt)
      (else (uhoh "unpermitted operator")))))

(define ensure-number
  (lambda (e)
    (if (number? e) e
        (uhoh "non-number"))))

(define print-form
  (lambda ()
    (display "<form action=\"")
    (display (getenv "SCRIPT_NAME"))
    (display "\">
  Enter arithmetic expression:<br>
  <input type=textarea name=arithexp><p>
  <input type=submit value=\"Evaluate\">
  <input type=reset value=\"Clear\">
</form>")))

(define print-page-begin
  (lambda ()
    (display "content-type: text/html

<html>
  <head>
    <title>A Scheme Calculator</title>
  </head>
  <body>")))

(define print-page-end
  (lambda ()
    (display "</body>
</html>")))

(parse-form-data)

(print-page-begin)

(let ((e (form-data-get "arithexp")))
  (unless (null? e)
    (let ((e1 (car e)))
      (display-html e1)
      (display "<p>
```

```
  =&gt;  ")
      (display-html
       (call/cc
        (lambda (k)
          (set! uhoh
                (lambda (s)
                  (k (string-append "Error: " s))))
          (number->string
           (calc-eval (read (open-input-string (car e))))))))
      (display "<p>"))))

(print-form)
(print-page-end)
```

附录 A Scheme方言

所有主要的Scheme方言都实现了R5RS规范。如果只使用R5RS中规定的功能，我们就能写出在这些方言中都能正常运行的代码。然而R5RS可能是为了更好的统一性，或是由于不可避免的系统依赖，在一些通用编程中无法忽略的重要问题上没有给出标准。因此这些Scheme方言不得不用一种特殊的非标准手段来解决这些问题。

本书使用了Scheme的MzScheme方言，因此也使用了一些非标准的特性。以下是本书中所有非标准的、依赖于MzScheme提供的特性：

- 命令行（包括打开一个侦听会话以及Shell脚本）
- `define-macro`
- `delete-file`
- `file-exists?`
- `file-or-directory-modify-seconds`
- `fluid-let`
- `gensym`
- `getenv`
- `get-output-string`
- `load-relative`
- `open-input-string`
- `open-output-string`
- `read-line`
- `reverse!`
- `system`
- `unless`
- `when`

以上这些命令中除了 `define-macro` 和 `system` 外都是在MzScheme的默认环境中就有的。而这两个缺少的则可以在MzScheme的标准库中找到，通过以下方式显式地加载。

```
(require (lib "defmacro.ss")) ;provides define-macro
(require (lib "process.ss"))  ;provides system
```

另外还可以把这两段代码放在MzScheme的初始化文件中（在Unix系统下是用户家目录下的 `.mzschemerc` 文件）。

一些非标准的特性（如 `file-exists?` 和 `delete-file` ）事实上在很多Scheme实现中已经是“标准”的特性了。另一些特性（如 `when` 和 `unless` ）或多或少有种“插件”式的定义（在本书中给出），因此可以在任何不具备这些过程的Scheme中加载。其他的需要针对每种方言来定义（如 `load-relative` ）。

本章描述了如何给你使用的Scheme方言加上本书中用到的这些非标准特性。想要了解更多关于你使用的Scheme方言，请参考其实现者提供的文档（附录E）。

A.1 调用和初始化文件

很多Scheme方言就像MzScheme一样都会从用户的家目录中载入初始化文件。我们可以把非标准功能的定义都放到这个初始化文件中，这样非常方便。比如，非标准过程 `file-or-directory-modify-seconds` 可以添加到Guile语言中，只要把下面的代码放到Guile的初始化文件（ `~/.guile` ）中即可：

```
(define file-or-directory-modify-seconds
  (lambda (f)
    (vector-ref (stat f) 9)))
```

另外，不同的Scheme方言有他们自己的不同的命令来启动对应的侦听器。下面的表格列出了不同Scheme方言对应的启动命令和初始化文件位置：

Dialect name	*Command*	*Init file*
Bigloo	`bigloo`	`~/.bigloorc`
Chicken	`csi`	`~/.csirc`
Gambit	`gsi`	`~/gambc.scm`
Gauche	`gosh`	`~/.gaucherc`
Guile	`guile`	`~/.guile`
Kawa	`kawa`	`~/.kawarc.scm`
MIT Scheme (Unix)	`scheme`	`~/.scheme.init`
MIT Scheme (Win)	`scheme`	`~/scheme.ini`
MzScheme (Unix, Mac OS X)	`mzscheme`	`~/.mzschemerc`
MzScheme (Win, Mac OS Classic)	`mzscheme`	`~/mzschemerc.ss`
SCM	`scm`	`~/ScmInit.scm`
STk	`snow`	`~/.stkrc`

A.2 Shell脚本

使用Guile编写的Shell脚本的初始行差不多应该是：

```
":";exec guile -s $0 "$@"
```

在Guile脚本中，调用过程 `(command-line)` 会以列表的形式返回脚本的名称和参数。如果只需要参数，只需要获得列表的 `cdr` 部分即可。

用Gauche编写的Shell脚本以：

```
":"; exec gosh -- $0 "$@"
```

开头。在脚本中变量 `*argv*` 中保存着脚本的参数列表。

用SCM编写的Shell脚本以：

```
":";exec scm -l $0 "$@"
```

开头。脚本中变量 `*argv*` 保存着一个列表，列表中包括Scheme可执行文件的名称，脚本的名称， `-l` 这个选项，还有脚本的参数。如果只需要参数，对列表执行 `cdddr` 即可。

STk的Shell脚本以：

```
":";exec snow -f $0 "$@"
```

开头。在脚本中变量 `*argv*` 中保存着脚本的参数列表。

A.3 `define-macro`

本文中使用的 `define-macro` 宏在Scheme的很多方言如Bigloo，Chicken，Gambit，Gauche，Guile，MzScheme和Pocket中都有定义。在其他Scheme方言中定义宏的方式基本上是相同的。本节将指出其他Scheme方言是如何表示如下一段代码片段的：

```
(define-macro MACRO-NAME
  (lambda MACRO-ARGS
    MACRO-BODY ...))
```

在MIT Scheme第7.7.1或更高版本中，上述代码被写为：

```
(define-syntax MACRO-NAME
  (rsc-macro-transformer
    (let ((xfmr (lambda MACRO-ARGS MACRO-BODY ...)))
      (lambda (e r)
        (apply xfmr (cdr e))))))
```

在老版本的MIT Scheme中：

```
(syntax-table-define system-global-syntax-table 'MACRO-NAME
  (macro MACRO-ARGS
    MACRO-BODY ...))
```

在SCM和Kawa中：

```
(defmacro MACRO-NAME MACRO-ARGS
  MACRO-BODY ...)
```

在STk中：

```
(define-macro (MACRO-NAME . MACRO-ARGS)
  MACRO-BODY ...)
```

A.4 `load-relative`

过程 `load-relative` 可以在Guile中如下定义：

```
(define load-relative
  (lambda (f)
    (let* ((n (string-length f))
           (full-pathname?
             (and (> n 0)
                  (let ((c0 (string-ref f 0)))
                    (or (char=? c0 #\/)
                        (char=? c0 #\~))))))
      (basic-load
        (if full-pathname? f
            (let ((clp (current-load-port)))
              (if clp
                  (string-append
                    (dirname (port-filename clp)) "/" f)
                  f)))))))
```

在SCM中可以这样写：

```
(define load-relative
  (lambda (f)
    (let* ((n (string-length f))
           (full-pathname?
            (and (> n 0)
                 (let ((c0 (string-ref f 0)))
                   (or (char=? c0 #\/)
                       (char=? c0 #\~))))))
    (load (if (and *load-pathname* full-pathname?)
              (in-vicinity (program-vicinity) f)
              f)))))
```

对于STk，下面的 `load-relative` 过程仅在没有使用 `load` 过程时生效：

```
(define *load-pathname* #f)

(define stk%load load)

(define load-relative
  (lambda (f)
    (fluid-let ((*load-pathname*
                 (if (not *load-pathname*) f
                     (let* ((n (string-length f))
                            (full-pathname?
                             (and (> n 0)
                                  (let ((c0 (string-ref f 0)))
                                    (or (char=? c0 #\/)
                                        (char=? c0 #\~))))))
                       (if full-pathname? f
                           (string-append
                             (dirname *load-pathname*)
                             "/" f))))))
      (stk%load *load-pathname*))))

(define load
  (lambda (f)
    (error "Don't use load.  Use load-relative instead.")))
```

我们使用 `~/filename` 表示用户家目录中被调用的文件。

附录B 在DOS中运行Scheme脚本

DOS的脚本也叫做“批处理”。通常一个输出 `Hello World!` 的DOS批处理文件应该这样写：

```
echo Hello, World!
```

这里用到了DOS的命令 `echo` 。脚本文件被命名为 `hello.bat` ，这样会被操作系统认为是可执行的。然后可以放入任一在 `PATH` 环境变量中的目录下。然后任何时候在DOS提示符下输入

```
hello.bat
```

或者更简单的：

```
hello
```

就能立即得到这句俗得不能再俗的问候。

Scheme版本的 `hello` 批处理文件会用Scheme产生相同的输出，但是我们需要在文件中写一些东西来告诉DOS让它用Scheme来分析文件内容，而不是理解为它默认的脚本批处理语言。Scheme的批处理文件（也命名为 `hello.bat` ），内容如下：

```
;@echo off
;goto :start
#|
:start
echo. > c:\_temp.scm
echo (load (find-executable-path "hello.bat" >> c:\_temp.scm
echo "hello.bat")) >> c:\_temp.scm
mzscheme -r c:\_temp.scm %1 %2 %3 %4 %5 %6 %7 %8 %9
goto :eof
|#

(display "Hello, World!")
(newline)

;:eof
```

到 `|#` 之前全部是标准的DOS批处理命令。后面是Scheme的问候代码。最后还有一行标准的DOS批处理，即 `;:eof` 。

当用户在DOS提示符下输入 `hello` 时，DOS读取并将 `hello.bat` 作为一个普通的批处理文件来运行。第一行， `;@echo off` ，关闭了命令运行时产生的输出——我们不想让大量冗余信息淹没我们脚本产生的输出。第二行， `;goto :start` ，让脚本的执行跳转到标号 `:start` 即第四行。后面接着的 `echo` 行创建了一个叫 `c:\_temp.scm` 的Scheme临时文件，其内容如下：

```
(load (find-executable-path "hello.bat" "hello.bat"))
```

下一个批处理命令调用MzScheme。 `-r` 选项加载Scheme文件 `c:\_temp.scm` 。所有的参数（在这个例子里没有）可以在Scheme中通过 `argv` 向量来获得。这个调用的Scheme会对我们的脚本进行求值，我们下面会看到。Scheme执行返回后，我们仍然需要让批处理文件正常地结束（否则就会碰到它不认识的Scheme代码了）。下一个批处理命令是 `goto :eof` ，这会让控制流跳过所有的Scheme代码，到达文件末尾，也就是包含 `;:eof` 标签的一行。然后脚本结束运行。

现在我们知道如何让Scheme来执行它的那部分代码，即运行嵌入在批处理文件中的Scheme表达式。载入 `c:\_temp.scm` 会使Scheme找到 `hello.bat` 文件的绝对路径（用 `find-executable-path` 过程），然后载入 `hello.bat` 。

因此，Scheme脚本文件现在会以Scheme文件来运行，文件中的Scheme表达式可以通过向量 `argv` 来访问脚本的原始参数。

现在Scheme略过脚本中的批处理命令。这是因为这些批处理命令要么以分号开头，要么用 `#|...|#` 包裹，这在Scheme看来是注释。

文件剩下的部分当然是Scheme代码，因此表达式被依次求值（最后的表达式 `;:eof` 是一个Scheme注释，因此没有关系）总之所有的表达式被求值后，Scheme会退出。

综上所述，在DOS提示符下输入hello会产生

```
Hello, World!
```

并返回DOS提示符。

附录 C 数值计算技术

递归（包括循环）与Scheme的算术基本过程结合可以实现各种数值计算技术。作为一个例子，我们来实现辛普森法则，这是一个用来计算定积分的数值解的过程。

C.1 辛普森积分法

函数f(x)在区间[a,b]上的定积分可以看作是f(x)曲线下方从x=a到x=b的区域的面积。也就是说，我们把f(x)的曲线绘制在xy平面上，然后找到由该曲线，x轴，x=a和x=b所围成区域的面积即是积分值。



根据辛普森法则，我们把积分区间[a,b]划分为n个相等的区间，n是一个偶数（n越大，近似的效果就越好）。区间边界在x轴上形成了n+1个点，即：$x0, x_1, \ldots, x_i, x{i+1}, \ldots, x_n$ ，其中 $x_0=a, x_n=b$ 。每个小区间的长度是h=(b-a)/n，这样 $x_i=a+ih$ ，我们然后计算f(x)在区间端点的纵坐标值，即，其中 $y_i=f(x_i)=f(x+ih)$。辛普森法则用下列算式模拟f(x)在a到b之间的定积分：

$$\frac{h}{3}[(y0+y_n)+4(y_1+y_3+\dots +y{n-1})+2(y2+y_4+\dots +y{n-2})]$$

我们定义一个过程 `integrate-simpson` ，该过程接受四个参数：积分函数 `f` ，积分限 `a` 和 `b` ，以及划分区间的数目 `n` 。

```
(define integrate-simpson
  (lambda (f a b n)
    ;...
```

首先我们需要在 `integrate-simpson` 函数里做的是保证 `n` 为偶数，如果不是的话，我们直接把 `n` 加上1.

```
    ;...
    (unless (even? n) (set! n (+ n 1)))
    ;...
```

接下来我们把区间长度保存在变量 `h` 中。我们引入两个变量 `h*2` 和 `n/2` 来保存 `h` 的两倍和 `n` 的一半，因为我们会在后面的计算过程中经常用到这两个变量。

```
    ;...
    (let* ((h (/ (- b a) n))
           (h*2 (* h 2))
           (n/2 (/ n 2))
           ;...
```

我们注意到 $y1 + y_3 + \cdots + y{n−1}$ 与 $y2 + y_4 + \cdots + y{n−2}$ 都要把每个纵坐标进行相加。所以我们定义一个本地过程 `sum-every-other-ordinate-starting-from` 来捕获公共的循环过程。通过把这个循环抽象为一个过程，我们可以避免重复写这个循环。这样不仅减少了劳动量，而且减少了错误发生的可能，因为我们只需要调试一个地方即可。

过程 `sum-every-other-ordinate-starting-from` 接受两个参数：起始纵坐标和被相加的纵坐标的数量。

```
           ;...
           (sum-every-other-ordinate-starting-from
             (lambda (x0 num-ordinates)
               (let loop ((x x0) (i 0) (r 0))
                 (if (>= i num-ordinates) r
                     (loop (+ x h*2)
                           (+ i 1)
                           (+ r (f x)))))))
           ;...
```

我们现在可以计算着三个纵坐标的和，然后把它们拼起来得到最后的结果。注意 $y1+y_3+ \cdots + y{n−1}$中有n/2项，在 $y2 + y_4 + \cdots + y{n−2}$ 中有(n/2)-1项。

```
           ;...
           (y0+yn (+ (f a) (f b)))
           (y1+y3+...+y.n-1
             (sum-every-other-ordinate-starting-from
               (+ a h) n/2))
           (y2+y4+...+y.n-2
             (sum-every-other-ordinate-starting-from
               (+ a h*2) (- n/2 1))))
      (* 1/3 h
         (+ y0+yn
            (* 4.0 y1+y3+...+y.n-1)
            (* 2.0 y2+y4+...+y.n-2))))))
```

现在我们来用 `integrate-simpson` 来求下面函数的定积分：

$$\phi(x) = \frac{1}{\sqrt{2\pi}}e^{-\frac{x^2}{2}}$$

我们首先需要用Scheme的S表达式来定义 ϕ 。

```
(define *pi* (* 4 (atan 1)))

(define phi
  (lambda (x)
    (* (/ 1 (sqrt (* 2 *pi*)))
       (exp (- (* 1/2 (* x x)))))))
```

注意我们用 $tan^{-1}1 = \frac{\pi}{4}$ 来定义 `*pi*` 。

下面的调用分别计算了从0到1,2,3的积分值。都使用了10个区间。

```
(integrate-simpson phi 0 1 10)
(integrate-simpson phi 0 2 10)
(integrate-simpson phi 0 3 10)
```

如果精确到小数点后四位，上面的值应该分别是0.3413 0.4772 0.4987。可以看出我们实现的辛普森积分法确实获得了相当精确的值！

C.2 自适应区间长度

每次都指定区间数目感觉不是很方便。对某个积分来说足够好的 `n` 可能对另一个积分来说差太多。这种情况下，最好指定一个可以接受的误差 `e` ，然后让程序计算到底需要分多少个区间。完成该任务的典型方法是让程序通过增加n来得到更好的结果，直到连续两次结果之间的误差小于 `e` 。因此：

```
(define integrate-adaptive-simpson-first-try
  (lambda (f a b e)
    (let loop ((n 4)
               (iprev (integrate-simpson f a b 2)))
      (let ((icurr (integrate-simpson f a b n)))
        (if (<= (abs (- icurr iprev)) e)
            icurr
            (loop (+ n 2)))))))
```

这里我们连续两次计算辛普森积分（用我们最初定义的过程 `integrate-simpson` ）， `n` 从2,4，。。。注意 `n` 必须是偶数。当当前 `n` 的积分值 `icurr` 与前一次 `n` 的积分值 `iprev` 的差小于 `e` 时，我们返回 `icurr` 。

这种方法的问题是我们没有考虑对于某个函数来说可能只有某一段或多段能从增长的区间中获益。而对于函数的其他段而言，区间增长只会增加计算量，而不会让整体的结果更好。对于一个增长的适应过程而言，我们可以把积分拆成相邻的几段，让每段的精度独立的增长。

```
(define integrate-adaptive-simpson-second-try
  (lambda (f a b e)
    (let integrate-segment ((a a) (b b) (e e))
      (let ((i2 (integrate-simpson f a b 2))
            (i4 (integrate-simpson f a b 4)))
        (if (<= (abs (- i2 i4)) e)
            i4
            (let ((c (/ (+ a b) 2))
                  (e (/ e 2)))
              (+ (integrate-segment a c e)
                 (integrate-segment c b e))))))))
```

初始段是从 `a` 到 `b` ，为了找到一段的积分，我们用两个最小的区间数目2和4来计算辛普森积分 `i2` 和 `i4` 。如果误差在 `e` 以内，就返回 `i4` 。如果没有的话我们就把区间分成两份，递归计算每段的积分并相加。通常，同一层的不同段以它们自己的节奏来汇聚。注意当我们积分半段时，允许的误差也要减半，这样才不会产生精度丢失。

这个过程中仍然存在一些低效之处：积分 `i4` 重新计算了计算 `i2` 时用到的三个纵坐标，而且每个半段的积分都重新计算了 `i2` 和 `i4` 用到的三个纵坐标。我们可以通过显式保存 `i2` 和 `i4` 用到的和并在命名let中传更多参数来解决这种低效率。这样有利于在 `integrate-segment` 内部和连续调用 `integrate-segment` 时共享一些信息：

```
(define integrate-adaptive-simpson
  (lambda (f a b e)
    (let* ((h (/ (- b a) 4))
           (mid.a.b (+ a (* 2 h))))
      (let integrate-segment ((x0 a)
                              (x2 mid.a.b)
                              (x4 b)
                              (y0 (f a))
                              (y2 (f mid.a.b))
                              (y4 (f b))
                              (h h)
                              (e e))
        (let* ((x1 (+ x0 h))
               (x3 (+ x2 h))
               (y1 (f x1))
               (y3 (f x3))
               (i2 (* 2/3 h (+ y0 y4 (* 4.0 y2))))
               (i4 (* 1/3 h (+ y0 y4 (* 4.0 (+ y1 y3))
                                     (* 2.0 y2)))))
          (if (<= (abs (- i2 i4)) e)
              i4
              (let ((h (/ h 2)) (e (/ e 2)))
                (+ (integrate-segment
                     x0 x1 x2 y0 y1 y2 h e)
                   (integrate-segment
                     x2 x3 x4 y2 y3 y4 h e)))))))))
```

`integrate-segment` 现在显式地设置了四个 `h` 大小的区间，五个纵坐标 `y0`，`y1`，`y2`，`y3`，`y4` 。积分 `i4` 用到了所有的坐标值， `i2` 的区间大小是两倍的 `h` ，故只用到了 `y0`，`y2`，`y4` 。很容易看出 `i2` 和 `i4` 用到的和符合辛普森公式中的和。

比较下面对积分 $\int_{0}^{20}e^{x}dx$ 的近似：

```
(integrate-simpson          exp 0 20 10)
(integrate-simpson          exp 0 20 20)
(integrate-simpson          exp 0 20 40)
(integrate-adaptive-simpson exp 0 20 .001)
(- (exp 20) 1)
```

可以分析出最后一个是正确的答案。看看你能不能找到一个最小的 `n` （如果设得太小会算得超级慢。。。）让 `(integrate-simpson exp 0 20 n)` 返回一个和 `integrate-adaptive-simpson` 算出的差不多的答案？

C.3 广义积分（反常积分）

辛普森积分法不能直接用来计算广义积分（这种积分的被积函数在某个点的值无穷大或者积分区间的端点无穷大）。然而这个积分法仍然可以用于部分积分，而剩下的部分用其他办法来获得近似值。比如，考虑伽玛函数 Γ 。对n>0，$\Gamma(n)$ 被定义为下面的积分（积分上限为无穷）：

$$\Gamma(n)=\int_{0}^{\infty} x^{n-1}e^{-x}dx$$

从上式可以看出两个结论：

a. $\Gamma(1)=1$ b. 对n>0， $\Gamma(n+1)=n\Gamma(n)$

这就意味着如果我们知道 Γ 在区间(1,2)上的值，我们就可以知道任何n>0 的 $\Gamma(n)$ 。实际上，如果我们放宽条件n>0，我们可以用结论b来把 $\Gamma(n)$ 扩展到n ≤ 0，而函数在n ≤ 0时会发散。

我们首先实现一个Scheme过程 `gamma-1-to-2` ，其参数n在区间(1,2)内。 `gamma-1-to-2` 的第二个参数 `e` 是精确度。

```
(define gamma-1-to-2
  (lambda (n e)
    (unless (< 1 n 2)
      (error 'gamma-1-to-2 "argument outside (1, 2)"))
    ;...
```

我们引入一个局部变量 `gamma-integrand` 来保存 Γ 中的被积函数 $g(x)=x^{n-1}e^x$ ：

```
    ;...
    (let ((gamma-integrand
            (let ((n-1 (- n 1)))
              (lambda (x)
                (* (expt x n-1)
                   (exp (- x))))))
          ;...
```

我们现在需要让g(x)从0积分到 ∞ ，首先我们没法定义一个“无穷”的区间表示；因此我们用辛普森公式只积分一个部分，如 $[0,x_c]$ （c的意思是截取(cut)），对于剩下的部位，“尾部”，区间 $[x_c,\infty]$ ，我们用一个“尾部”被积函数t(x)来近似g(x)，这样更加容易分析和处理。事实上，可以很容易看出对于足够大的 x_c ，我们可以把g(x)替换为一个递减的指数函数 $t(x)=y_ce^{-(x-x_c)}$ ，其中 $y_c=g(x_c)$ ，因此：

$$\int0^{\infty}g(x)dx \approx \int_0^{x_c}g(x)dx + \int{x_c}^{\infty}t(x)dx$$

前一个积分可以用辛普森公式来解出，后一个就是 y_c 。为了找 x_c ，我们从一个比较小的值（如 `4` ）开始，然后通过每次扩大一倍来改进积分结果，直到在 $2x_c$ 处的纵坐标（即 $g(2x_c)$ ）在一个特定的由“尾部”的被积函数纵坐标

误差之内。对于辛普森积分和尾部的积分计算，我们都需要误差为 `e/100` ，就是比 `e` 再小两个数量级，这样对总体的误差不会有太大影响：

```
          ;...
          (e/100 (/ e 100)))
       (let loop ((xc 4) (yc (gamma-integrand 4)))
         (let* ((tail-integrand
                  (lambda (x)
                    (* yc (exp (- (- x xc))))))
                (x1 (* 2 xc))
                (y1 (gamma-integrand x1))
                (y1-estimated (tail-integrand x1)))
           (if (<= (abs (- y1 y1-estimated)) e/100)
               (+ (integrate-adaptive-simpson
                    gamma-integrand
                    0 xc e/100)
                  yc)
               (loop x1 y1)))))))
```

我们现在可以写一个更通用的过程 `gamma` 来返回任意 `n` 对应的 $\Gamma(n)$ ：

```
(define gamma
  (lambda (n e)
    (cond ((< n 1) (/ (gamma (+ n 1) e) n))
          ((= n 1) 1)
          ((< 1 n 2) (gamma-1-to-2 n e))
          (else (let ((n-1 (- n 1)))
                  (* n-1 (gamma n-1 e)))))))
```

我们现在来计算 $\Gamma(\frac{3}{2})$ ：

```
(gamma 3/2 .001)
(* 1/2 (sqrt *pi*))
```

第二个值是理论上的正确答案。（这是由于 $\Gamma(\frac{3}{2})=\frac{1}{2}\Gamma(\frac{1}{2})$ ，而可以证明 $\Gamma(\frac{1}{2})$ 的值是 $\pi^{\frac{1}{2}}$ ）你可以通过更改过程 `gamma` 的第二个参数（误差）让结果达到任何你想要的近似程度。

---

[1]. 想了解为什么这种近似是合理的，请参考任意一种基础的微积分教程。 [2]. ϕ 是服从正态或高斯分布的随机变量的概率密度函数。其均值为0而方差为1。定积分 $\int_0^z\phi(x)dx $ 是该随机变量在0到z之间取值的概率。然而你并不需要了解这么多也可以理解这个例子！ [3]. 如果Scheme没有 `atan` 过程，我们可以用我们的数值积分过程来得到积分 $\int_0^1(1+x^2)^{-1}dx$ 的值，即 $\pi/4$。 [4].

把常量——如 `phi` 中的 `(/ 1 (sqrt (* 2 *pi*)))` ——提取到被积分函数外面，可以加速 `integrate-simpson` 中纵坐标的计算。 [5]. 对大于0的实数n来说$\Gamma(n)$是“减小后阶乘”函数（把正整数n映射到(n-1)!）的一个扩展。

附录D 可设为infinity的时钟

Guile的过程 `alarm` 提供了一种可中断的定时器机制。用户可以给这个时钟设置或重置一些时间片，或者让它停止。当时钟的定时器递减到0后，它就会执行用户之前设定的动作。Guile的 `alarm` 不是一个类似于第十五章第一节里定义的那种时钟，但是我们可以很容易的把它改造成那样。

时钟的定时器的初始状态是停止的，也就是说它不会随着时间流逝而被“触发”。如果想把定时器的触发时间设置为 `n` 秒（ `n` 不为0），运行 `(alarm n)` 。如果定时器已经被设定过了，那么 `(alarm n)` 就返回该定时器在本次设定前剩余的秒数。如果之前没有设定过，则返回0。

执行 `(alarm 0)` 让时钟的定时器停止，即定时器中的计数器【你可以理解为一个变量】不会随时间而递减，而且不会触发。 `(alarm 0)` 同样返回定时器在本次设定前剩余的秒数（如果之前设定过的话）。

默认情况下，当时钟的定时器计数减到0时，Guile会在控制台上显示一条消息并退出。更多的行为可以用过程 `sigaction` 来设定，如下所示：

```
(sigaction SIGALRM
  (lambda (sig)
    (display "Signal ")
    (display sig)
    (display " raised.  Continuing...")
    (newline)))
```

第一个参数 `SIGALRM` （恰好是14）告诉 `sigaction` 需要设定的时钟处理函数[1]。第二个参数是一个用户指定的单参数过程。在这个例子里，当时钟触发时，处理函数会在控制台上显示 `"Signal 14 raised. Continuing..."` 而不是退出Scheme（14是变量 `SIGALRM` 的值，时钟会把它传递给它对应的处理过程，我们现在先不考虑这个）。

从我们的角度看，这种简单的定时器机制有一个问题。过程 `alarm` 的返回值 `0` 的意义是不明确的：既可能是指时钟处于停止状态，也有可能是刚好计时器减到了0。我们可以通过在时钟的算法里引入 `*infinity*` 来解决这个问题。换句话说，我们需要的时钟与 `alarm` 基本上是差不多的，除了一点，那就是如果时钟停止的话，那么它有 `*infinity*` 秒。这样就看起来比较自然了。

1. `(clock n)` 对于一个停止的时钟返回 `*infinity*` ，而不是0。
2. 如果让时钟停止，执行 `(clock *infinity*)` ，而不是 `(clock 0)` 。
3. `(clock 0)` 相当于给时钟设置一个无限小的时间，也就是让它立即触发。

在Guile中，我们可以把 `*infinity*` 定义为如下的“数”：

```
(define *infinity* (/ 1 0))
```

我们用 `alarm` 来定义 `clock` 。

```
(define clock
  (let ((stopped? #t)
        (clock-interrupt-handler
         (lambda () (error "Clock interrupt!"))))
    (let ((generate-clock-interrupt
           (lambda ()
             (set! stopped? #t)
             (clock-interrupt-handler))))
      (sigaction SIGALRM
                 (lambda (sig) (generate-clock-interrupt)))
      (lambda (msg val)
        (case msg
          ((set-handler)
           (set! clock-interrupt-handler val))
          ((set)
           (cond ((= val *infinity*)
                  ;This is equivalent to stopping the clock.
                  ;This is almost equivalent to (alarm 0), except
                  ;that if the clock is already stopped,
                  ;return *infinity*.

                  (let ((time-remaining (alarm 0)))
                    (if stopped? *infinity*
                        (begin (set! stopped? #t)
                          time-remaining))))

                 ((= val 0)
                  ;This is equivalent to setting the alarm to
                  ;go off immediately.  This is almost equivalent
                  ;to (alarm 0), except you force the alarm
                  ;handler to run.

                  (let ((time-remaining (alarm 0)))
                    (if stopped?
                        (begin (generate-clock-interrupt)
                          *infinity*)
                        (begin (generate-clock-interrupt)
                          time-remaining))))

                 (else
                  ;This is equivalent to (alarm n) for n != 0.
                  ;Just remember to return *infinity* if the
                  ;clock was previously quiescent.

                  (let ((time-remaining (alarm val)))
                    (if stopped?
                        (begin (set! stopped? #f) *infinity*)
                        time-remaining))))))))))
```

过程`clock`用到了三个内部状态变量：

1. `stopped?`，表示时钟是否是停止的；
2. `clock-interrupt-handler`，一个过程，表示用户希望在时钟触发后执行的动作；
3. `generate-clock-interrupt`，另一个过程，该过程会在运行用户定义的时钟处理过程前把`stopped?`设为`false`。

过程`clock`有两个参数。如果第一个参数是`set-handler`，那么就把第二个参数作为时钟处理器。

如果第一个参数是`set`，就把该时钟触发时间设置为第二个参数，返回本次设定前定时器剩余的秒数。代码对`0`、`*infinity*`以及其他时间值的处理是不同的，这样用户可以得到一个算术上对`alarm`透明的接口。

---

[1]. 还有一些其他的信号和与之相应的处理器，`sigaction`同样可以使用它们。

附录E 参考文献

译注：文献名称用斜体标出

[1]	Harold Abelson and Gerald?Jay Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs ("SICP")*. MIT Press, 2nd edition, 1996.
[2]	Milton Abramowitz and Irene?A Stegun, editors. *Handbook of Mathematical Functions: with Formulas, Graphs, and Mathematical Tables*. Dover Publications, 1965.
[3]	Per Bothner. The Kawa Scheme system.
[4]	William Clinger. *Nondeterministic call by need is neither lazy nor by name*. In Proc ACM Symp Lisp and Functional Programming, pages 226–234, 1982.
[5]	R?Kent Dybvig. *The Scheme Programming Language*. Prentice Hall PTR, 2nd edition, 1996.
[6]	Marc Feeley. Gambit Scheme System.
[7]	Matthias Felleisen. *Transliterating Prolog into Scheme*. Technical Report 182, Indiana U Comp Sci Dept, 1985.
[8]	Matthias Felleisen, Robert?Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. *How to Design Programs: An Introduction to Programming and Computing*. MIT Press, 2001.
[9]	Matthew Flatt. MzScheme.
[10]	Daniel?P Friedman and Matthias Felleisen. *The Little Schemer*. MIT Press, 4th edition, 1996.
[11]	Daniel?P Friedman and Matthias Felleisen. *The Seasoned Schemer*. MIT Press, 1996.
[12]	Daniel?P Friedman, Mitchell Wand, and Christopher?T Haynes. *Essentials of Programming Languages*. MIT Press, McGraw-Hill, 1992.
[13]	FSF. Guile: Project GNU's Extension Language.
[14]	Erick Gallesio. STk.
[15]	Ben Goetter. Pocket Scheme for the H/PC and P/PC.
[16]	Christopher?T Haynes. *Logic continuations*. In J Logic Program, pages 157–176, 1987. vol 4.

[17]	Christopher?T Haynes and Daniel?P Friedman. *Engines Build Process Abstractions*. In Conf ACM Symp Lisp and Functional Programming, pages 18–24, 1984.
[18]	Christopher?T Haynes, Daniel?P Friedman, and Mitchell Wand. *Continuations and Coroutines*. In Conf ACM Symp Lisp and Functional Programming, pages 293–298, 1984.
[19]	J?A?H Hunter. *Mathematical Brain-Teasers*. Dover Publications, 1976.
[20]	Aubrey Jaffer. SCM.
[21]	Shiro Kawai. Gauche: A Scheme Implementation.
[22]	Sonya?E Keene. *Object-oriented Programming in Common Lisp: A Programmer's Guide to CLOS*. Addison-Wesley, 1989.
[23]	Richard Kelsey, William Clinger, and Jonathan Rees (eds). *Revised?5 Report on the Algorithmic Language Scheme (R5RS)*, 1998.
[24]	Gregor Kiczales, Jim des Rivie?res, and Daniel?G Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.
[25]	John McCarthy. *A Basis for a Mathematical Theory of Computation*. In P?Braffort and D?Hirschberg, editors, Computer Programming and Formal Systems. North-Holland, 1967.
[26]	MIT Scheme Team. MIT Scheme.
[27]	NCSA. The Common Gateway Interface.
[28]	Christian Queinnec. *Lisp in Small Pieces*. Cambridge University Press, 1996.
[29]	Thomas?L Saaty and Paul?C Kainen. *The Four-Color Problem: Assaults and Conquest*. Dover Publications, 1986.
[30]	Manuel Serrano. Bigloo.
[31]	Leon Sterling and Ehud Shapiro. *The Art of Prolog*. MIT Press, 2nd edition, 1994.
[32]	Felix?L Winkelmann. Chicken: A practical and portable Scheme system.
[33]	Ramin Zabih, David McAllester, and David Chapman. *Non-deterministic Lisp with dependency-directed backtracking*. In AAAI-87, pages 59–64, 1987.

附录F 索引

【译注】由于英文原HTML采用tex2page自动生成，故翻译时没有采用原HTML模板，因此有很多引用信息的丢失。所以目前中文版索引的精确度仅能定位到“章”，更精确的搜索请在浏览器上使用Ctrl-F搜索页面文本或查看英文原文。

Java 符号表设计的相关问题

翻译自：http://www.bearcave.com/software/java/java_symtab.html

很多Java语言处理器不会读Java，而是读Java类文件，并从类文件生成符号表和抽象语法树。Java类文件里的代码在语法和语义上都是正确的。结果就是这些工具的作者避免考虑实现一个Java前端时会遇到的很多困难的问题。

Java编程语言的设计者在设计这个语言时没有考虑实现的简单性。确实应当如此，因为更重要的是语言容易使用。设计Java编译器前端的语义分析时遇到的一个很困难的问题就是符号表的设计。这个页面零散地讨论了一些Java符号表设计时遇到的问题。

编译器的前端主要工作包括一下几点：

1. 解析源代码识别正确的程序，对不正确的结构报错。对BPI这个Java前端来说，这个工作由ANTLR生成的一个解析器完成。解析器的输出是一个抽象语法树（AST），包括了源代码里所有的声明。

2. 从Java类文件中读取声明信息，对于本地Java编译器来说，把AST编译为字节码。这也包括了下面的transitive closure（图中所有可以从根节点到达的节点，从这个角度讲这个图是一个类组成的树，通过这个树可以定义所有需要被编译器读取的类文件。

3. 处理AST和类文件中的声明，构造符号表。一旦这些声明节点被处理，就从AST当中剔除掉。

前端的输出是一个语法和语义上都正确的AST，每个节点都有一个指针指向一个标识符（如果这是一个叶节点的话）或一个类型（如果这是一个非终结节点或着一个类型引用，如MyType.class）。

“符号表”这个词通常指代一种比表格（比如struct组成的数组）数据结构。当符号和类型被解析的时候，符号表必须反应当前正在被处理的AST的作用域。比如，下面的C语言代码有三个叫 `x` 的变量，分散在不同的作用域里。

```
static char x;
  int foo() {
     int x;
     {
        float x;
     }
  }
```

解析符号和类型需要遍历AST来处理各种声明。在遍历AST中不同的作用域时，符号表始终反应当前作用域，这样在查找 `x` 的时候，当前作用域的符号会被返回。

符号表的作用域结构只在解析符号和类型时有用。名字一旦解析完成。AST中名字和它的符号的关系可以直接通过一个指针找到。

Pascal和C语言（这两种语言只有简单的分层作用域）的编译器使用的符号表通常都是直接镜像语言的作用域。每个作用域都有一个符号表。每个符号表都有一个指针指向它上层作用域。最上层的根符号表就是全局符号表，包含了全局的符号和函数（或者是Pascal里的过程）。当进入一个函数作用域时，就创建一个函数符号表。这个函数符号表的父指针指向前面紧接着的一个“上面的”层（或是全局符号表，或是Pascal里一个闭合的过程或函数）。一个块符号表指向它的父符号表，也就是函数符号表。符号搜索从当前作用域向全局作用域向上遍历进行。

一旦符号和类型解析完毕，作用域层级就不需要了。然而函数或是类的局部作用域仍然很重要，而且这些局部作用域必须仍然可以被编译器访问这个作用域里的所有符号。比如，为了在函数调用时分配堆栈，编译器必须能找到所有与这个方法相关的变量。Java编译器必须能购跟踪类的成员，因为这些变量会被分配到可以被垃圾回收的内存中。

大多数面向对象语言的作用域都比过程语言（C或PASCAL）要更复杂。C++支持多重继承，Java支持多接口定义（多重继承的一种正确方式）。符号表必须足够高效这样编译器前端才不会花大量时间在查找符号上。Java编译器的符号表设计主要有一下一些考虑：

1. Java有一个非常大的全局作用域，因为所有的类和包都被导入到这个全局的命名空间。全局符号必须存储在一个大容量的数据结构中，而且查找的时间复杂度是O(n)，比如一个哈希表。

2. Java有非常多的局部作用域（类，方法和块）只包含较少的符号（相对全局作用域而言），对于它们使用支持大容量和高速查找的数据结构有点过于复杂了（不论是内存使用还是代码复杂度）。局部作用域应该用一个简单而且相对快速（比如O(log2(n))）的数据结构实现。比如平衡树和跳跃列表。

3. 符号表应该能支持一个作用域里定义多个相同名字。符号表必须能帮助编译器解析两种相同类型（比如都是函数）的相同名字在同一作用域中多次声明产生的冲突。

在C语言里一个作用域里的名字必须是唯一的。比如，在C语言里一个叫MyType的类型和一个叫MyType的函数是不被允许的。在Java里一个作用域里的名字可以不是唯一的。名称会根据它所在的上下文来解析。比如：

```
class Rose {
  Rose( int val ) { juliette = val; }
  public int juliette;
} // Rose

class Venice {
  void thorn {
    garden = new Rose( 42 );
    Rose( 86 );
    garden.Rose( 94 );
  }

  Rose Rose( int val ) { garden.juliette = val; }
  Rose garden;
} // venice
```

这个例子中有一个名为Rose的类，一个名为Rose的构造函数，一个名为Rose的方法返回一个类型为Rose的对象。编译器必须要联系上下文才知道哪个是哪个。而且注意引用的Rose方法和garden类型是在引用后面声明的。

Java中大部分符号作用域可以被描述为一个简单的层次关系（低层有指向高层的指针），除了和Java类相关的接口列表。注意接口也可以从上次接口继承。下面是Java里作用域的分级：

```
Global (objects imported via import statements)
  Parent Interface (this may be a list)
    Interface (there may be a list of interfaces)
       Parent class
         Class
           Method
             Block
```

符号表和语义分析（检查Java解析器返回的AST）代码必须能够解析一个符号定义是否在语义上是正确的。一个名称的多个定义是允许的（比如多个类成员）。然而不明确的符号使用是不允许的：

> Java语言规范 (JLS) 8.3.3.3
>
> 一个类可以继承两个或更多相同名字的属性，或从两个接口继承或一个从父类继承一个从接口继承。只有在试图只用简称来模糊的引用时才会发生编译错误。明确的全称或带 `super` 关键字的属性访问是允许的。

父类和接口都可以把其中定义的符号导入本地作用域。下面的例子中符号 `x` 在 `bar` 和 `fu` 中都定义了，这是允许的，因为在 `DoD` 类中没有引用 `x` 。

```
interface bar {
  int x = 42;
}

class fu {
  double x;
}


class DoD extends fu implements bar {
  int y;  // No error, since there is no local reference to x
}
```

如果 `x` 在类 `DoD` 中被引用了，编译器必须报告一个错误，因为这种引用是不明确的

```
class DoD extends fu implements bar {
  int y;

  DoD() {
    y = x + 1;    // Error, since the reference to x is ambiguous
  }
}
```

简称的不明确性还会出现在接口定义的内部类和父类中：

```
interface BuildEmpire
{
  class KhubilaiKahn {
    public int a, b, c;
  }
}

class GengisKahn
{
  class KhubilaiKahn {
    public double x, y, z;
  }
}


class mongol extends GengisKahn implements BuildEmpire
{
  void mondo() {
    KhubilaiKahn TheKahn;  // Ambiguous reference to class Khubila
  }
}
```

Java不支持类的多重继承，但是允许一个类实现多个接口或一个接口扩展（继承）多个接口

> Java语言规范9.3
>
> 一个接口可以继承多个相同的名字，这种情况不会引起编译错误。然而在接口内部试图用简称来引用这个属性会导致编译错误，因为这样的引用是不明确的。

比如，在下面的代码中 `key` 是不明确的：

```
interface Maryland
{
  String key = "General William Odom";
}

interface ProcurementOffice
{
  String key = "Admiral Bobby Inman";
}


interface NoSuchAgency extends Maryland, ProcurementOffice
{
  String RealKey = key + "42"; // ambiguous reference to key
}
```

当当语义分析查找符号 `key` 时，符号表必须允许语义检查代码来决定有两个对 `key` 的定义。符号表必须对作用域里的符号分类（成员和成员在一起，类和类在一起）。不像有些符号（方法，类和成员变量）没有分类因为它们可以通过上下文区分。

一个方法的多次定义不会在Java中产生语义错误，因为没有多重继承。比如，如果一个同名方法从两个接口中继承，这个方法要么是相同的，要么是冗余版本。如果有一个本地方法和一个在父类中定义的方法有相同的名字和参数（签名）。本地方法会在一个“更低的”作用域并且覆盖父类的。

Java 符号表的实现

符号表需求

考虑以上讨论的几点，一个符号表必须满足以下需求：

1. 支持一个标识符的多种定义。
2. 在全局符号库中快速查找，时间复杂度O(n)
3. 在局部（类、方法和块）符号中相对快速的查找O(log2(n))

4. 支持Java的分层作用域
5. 可以按照符号类型搜索（成员、方法，类）
6. 快速决定一个符号定义是否是不明确的

符号的生存期

类似C的语言可以一次编译一个函数。全局符号表必须保留当前文件中函数和它们的参数的符号信息。但是其他局部符号信息可以在函数编译后忽略。当编译器处理完一个 `.c` 文件（和被它引用的文件）中所有的函数后，所有的符号都被忽略了。

C++可以用类似的方法来编译。定义在头文件中的类引用一个对象。当文件处理后所有的符号可以忽略了。

Java更复杂。Java编译器必须读取Java符号定义来构建Class树，这个树用来确定当前正在编译的类所引用的所有类文件。也就是包含 `main` 方法的对象。这点出发可以找到所有被引用的类。

理论上一旦所有引用的Java符号的类被编译后，这些符号就可以被忽略了。实际上这样造成如此多的问题还不如换一个内存大一点的系统。所以Java符号在整个编译期间都存在。

构建符号表作用域

符号表中分层的作用域只在语义分析时有用。分析结束后，所有的符号（标识符节点）都会指到正确的符号上。然而，一旦作用域构建完，它就在那里了。

每个局部作用域（块、方法和类）有一个局部的符号表指向包围它的符号表。在顶层是全局符号表包含所有全局类和导入的符号。进行语义分析时从局部符号表向上层搜索，搜索每个符号表直到全局符号表搜索完。如果搜完全局符号表还没有找到，这个符号就不存在。

Java的作用域不是一个由唯一的符号组成的简单分层结构（像C语言一样）。一个符号可能会有多个定义（类成员、方法或类名）。一个给定作用域的符号可能来自多个地方。比如下面的Java代码中类 `gin` 和接口 `tonic` 在同一层定义了相同的符号。

```
interface tonic {
    int water = 1;
    int quinine = 2;
    int sugar = 3;
    int TheSame = 4;
}

class gin {
    public int water, alcohol, juniper;
    public float TheSame;
}

class g_and_t extends gin implements tonic {
    class contextName {
      public int x, y, z;
    } // contextName

    public int contextName( int x ) { return x; }
    public contextName contextName;
}
```

作用域、局部变量、参数

Java里的局部变量是方法中的变量，这些变量被分配到一个由块或语句创建的堆栈中。如：

```
class bogus {
   public void foobar() {
      int a, b, c;

      { // this is a scope block
         int x, y, z;
      }
   }
}
```

不像C或C++，Java不允许重新声明局部变量：

> Java语言规范JLS 14.3.2
>
> 如果一个标识符被声明为局部变量，而在其作用域内已有一个参数或本地变量，编译器会报错。因此下面的例子无法通过编译：
>
> ```
> class Test {
>         public static void main( String[] args ) {
>           int i;
>
>           for (int i = 0; i < 10; i++)  // Error: local variabl
>           redeclared
>             System.out.println(i);
>         }
>       }
> ```

本地局部变量允许被重定义为类成员，这让变量重定义检查也成为语义分析的一部分工作。

向前引用

向前应用是引用一个声明写在该引用后面的符号。

当一个类属性被初始化时，初始器必须在前面已经声明并且初始化了。下面的例子（摘自JLS6.3）会报错：

```
class Test {
    int i = j;  // compile-time error: incorrect forward reference
    int j = 1;
}
```

本地局部变量也不能向前引用，如：

```
class geomancy {
   public float circleArea( float r ) {
      float area;

      area = pie * r * r;     // undefined variable 'pie'
      float pie = (float)Math.PI;

      return area;
   }
}
```

然而，向前引用允许从一个局部作用域（一个方法）引用一个在同一个类中定义的类成员。比如，在下面的Java代码中方法 `getHexChar` 向后引用了类成员 `hexTab` ：

```
class HexStuff {

  public char getHexChar( byte digit ) {

    digit = (byte)(digit & 0xf);
    char ch = hexTab[digit];  // legal forward reference to class r

    return ch;
  } // getHexchar

  private static char hexTab[] = new char[] {
    '0','1','2','3','4','5','6','7','8', '9', 'a', 'b', 'c', 'd',
  };

} // HexStuff
```

包

Java中最顶层编译单元是包，要么是一个显式命名的包或是一个未命名的包（包含 `main` 方法的函数）。所有的包都自动导入了默认的包 `java.lang.*` 和其他被本地系统所需要的包。用户可以显式的导入其他包。

当包A导入了包B，包B提供了：

- 用 `public` 修饰符标记的所有类和接口
- 子包（如导入到B中的其他包）

如果B包导入了包X，其中有一个公开类 `foo` ，这个类可以用全称 `X.foo` 引用。

包给符号表加入了另一层复杂度。一个包好比一个对象，该对象定义了一堆类，接口和子包。一旦包被编译器读取，在下面如果有相同的导入语句就不会再读了，因为它的定义编译器都已经知道了。

一个包所定义的类、接口和包被导入到当前包的全局作用域。在Java代码中，导入的包所定义的类名可以用简称来引用（JLS6.5.4），在被导入的包的子包中定义的类名可以用全称引用。然而在符号表中所有的类名都与一个全名关联着。

符号表实现概述

1. 支持一个给定标识符的多重定义。

所有有相同名字的标识符都被放在一个容器中。就像上面提到的，一个标识符可能是一个类成员，方法名或一个类名。一个定义可以有多个实例。比如上面Java代码中类成员 `TheSame` 有两个定义。容器可以用标识符的类型（成员，方法或类）来搜索，而且可以快速决定是否某个类型被多次定义（确定性引用）。如果一个对象命名了，符号会有一个属性指向它的上层（函数或类）。对于一个块这个指针是Null。注意上层不一定是上层作用域，定义在类 `gin` 和接口 `tonic` 中的符号在同一个作用域，但是它们有不同的上层。

2. 快速的全局搜索 全局符号表用大容量哈希表实现（哈希表能支持大量符号不用长的哈希链）

3. 包信息 一旦一个包被导入全局作用域，这个包就不再被引用了，导入的类名（类或接口）可以被引用，就如同它们是在当前编译单元中定义的（通过简称）。子包也成了全局作用域中的对象。包类型名和额外的子包可以用全称引用。 包定义保存在一个分开的包表里。包从这个表里导入到编译单元的全局作用域。包信息在整个编译期间都存在。

4. 局部查找 通常局部Java作用域中的符号很少。本地符号查找必须要快，但是不用像全局那么快，因为通常符号很少。 我设想了三种数据结构来实现局部符号表：

    - 跳跃列表（也可以查看Thomas Nienann关于跳跃列表的精彩网页
    - 红黑树（一种平衡二叉树）
    - 简单的二叉树 对于小的符号表这三种数据结构的搜索时间都差不多。二叉树在测试中是最小最简单的算法，所以选择它作为局部符号表。

5. 支持Java层次作用域 每个符号表都包含一个上层作用域的指针。

6. 支持以符号类型搜索 语义分析知道它所搜索符号的上下文（这个符号是成员、方法还是类）。符号表层次以标识符和类型来搜索。

7. 快速检测一个符号定义是否是模糊的 多个相同类型的符号定义（比如两个成员）被串在一起。如果 `next` 指针是NULL，那就有多个定义。错误报告代码可以用这些定义报告给用户冲突的符号是在哪里定义的。

符号表构造

在方法被处理之前，所有类成员引用都被处理并塞进符号表。这样在方法中对成员的引用就可以正确的解析了。

方法内的声明被顺序处理。如果函数中一个名字的引用不能“看到”，就报告一个错误（未定义的名称）。

递归编译和符号表

当一个编译单元（包）被编译时，所有它引用的类和包信息必须存在。《Java语言规范》没有准确定义这是怎么做的。规范中只说被编译的Java代码可以存在一个数据库里或在一个目录下，这个目录结构和包和类的全名一一对应。类和包必须可以访问。《Java虚拟机规范》定义了Java `.class` 字节码文件中的信息，但是没有

说编译顺序。尽管没有规范Java是如何编译的，但还是有“通用方法”。至少对于这个设计，“通用方法”基于Sun公司的 `javac` 编译器和微软的 `Visual J++` 编译器 `jvc` 。

当一个编译单元编译完成时，所有该编译单元所引用的外部类信息被记录在编译生成的字节码文件中。字节码文件可以打包成jar文件。就是一个用ZIP文件格式压缩存放的字节码文件层次。字节码或jar文件存放在当前文件夹或CLASSPATH环境变量指定的目录下。为了让这个机制工作。文件名最好和相关联的类名保持一致（如类 `FooBar` 用 `FooBar.java` 实现）

如果，当搜索类定义时，Java编译器只找到一个 `.java` 文件定义了这个类或者这个 `.java` 文件的时间戳比相应的字节码文件要新的话，Java编译器会重新编译这个类定义。

当编译顶层的编译单元时，Java编译器跟踪被导入到当前编译单元的包对象（一个包括了多个类和子包的包）。包中不是public的类定义不会被编译器保存，因为它们无法在包的外面看到。

Ian Kaplan, May 2, 2000 Revised most recently: May 31, 2000